AD-A214 433

(4)

The Rhetorical Knowledge Representation System: A User's Manual (For Rhet Version 15.25)

James F. Allen Bradford W. Miller

Technical Report 238 (rerevised)
March 1989

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

The Rhetorical Knowledge Representation System: A User's Manual (For Rhet Version 15.25)

James F. Allen Bradford W. Miller

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 238 (rerevised)

March 1989

This work was supported in part by ONR research contract no. N00014-80-C-0197, in part by U.S. Army Engineering Topographic Laboratories research contract no. DACA76-85-C-0001, and in part by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. (This contract supports the Northeast Artificial Intelligence Consortium (NAIC).)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 238 (rerevised)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Rhetorical Knowledge Representation System: A User's Manual (For the Rhet Version 15.25)		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) James F. Allen and Bradford W. Miller		8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0197 DACA76-85-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science 734 Computer Studies Bldg. Univ. of Rochester, Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA/1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE March 1989 13. NUMBER OF PAGES 219
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217 US Army ETL Fort Belvoir VA 22060		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution of this document is unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) knowledge representation, hybrid reasoning, frames, logic programming		

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Rhetorical (Rhet) is a programming/knowledge representation system that offers a set of tools for building an automated reasoning system. It's emphasis is on flexibility of representation, allowing the user to decide if the system will basically operate as a theorem prover, a frame-like system, or an associative network. Rhet may be used as the back-end to a user's programming system and handle the knowledge representation chores, or it may be used as a full-blown programming language.

Rhet offers two major modes of inference: a horn clause theorem prover

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (Continued)

(backwards chaining mechanism), and a forward chaining mechanism. Both modes use a common representation of facts, namely horn clauses with universally quantified, potentially type restricted, variables, and use the unification algorithm. Additionally, they both share the following additional specialized reasoning capabilities:

1. variables may be typed with a fairly general type theory that allows a limited calculus of types including intersection and subtraction;

2. full reasoning about equality between ground terms;

3. reasoning within a context space, with access to axioms and terms in parent contexts;

4. the user is allowed to specify specialized reasoners for determining if two objects of a special type will unify;

5. escapes in Lisp for use as necessary.

Rhet builds on some fairly recent work on PROLOG for efficiency, including such features as 'limited' intelligent backtracking.

Abstract

Rhetorical (Rhet) is a programming / knowledge representation system that offers a set of tools for building an automated reasoning system. It's emphasis is on flexibility of representation, allowing the user to decide if the system will basically operate as a theorem prover, a frame-like system, or an associative network. Rhet may be used as the back-end to a user's programming system and handle the knowledge representation chores, or it may be used as a full-blown programming language.

Rhet offers two major modes of inference: a horn clause theorem prover (backwards chaining mechanism), and a forward chaining mechanism. Both modes use a common representation of facts, namely horn clauses with universally quantified, potentially type restricted, variables, and use the unification algorithm. Additionally, they both share the following additional specialized reasoning capabilities:

1) variables may be typed with a fairly general type theory that allows a limited calculus of types including intersection and subtraction;

2) full reasoning about equality between ground terms;

3) reasoning within a context space, with access to axioms and terms in parent contexts;

4) the user is allowed to specify specialized reasoners for determining if two objects of a special type will unify;

5) escapes into Lisp for use as necessary.

Rhet builds on some fairly recent work on PROLOG for efficiency, including such features as 'limited' intelligent backtracking.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 What is this?

This documents the user interface for the Rhetorical[1] (Rhet) system. This system, functionally, is an extension to the HORNE[Allen and Miller, 1986] system we have been using at the University of Rochester for several years now, and has been leveraged off of our experience with the existing system.

Rhet is a horn clause based reasoning system that is embedded in a Common Lisp (CL) [Steele Jr., 1984] environment[2]. Its facilities are called as Lisp functions and Rhet programs can themselves call Lisp functions. Thus, effective programming in Rhet involves a careful mixture of logic programming and Lisp programming. This manual assumes that the user is familiar with the fundamentals of both Lisp and Prolog. The naive user

[1] Rochester Horn clause Extended Tool Of Research In Characterizing Applied natural Language

[2] Some manufacturer defined extensions are used, but they are minimal, for ease of rehosting between lispmachine type environments.

should consult one of [Winston and Horn, 1981], [Tatar, 1987], or [Wilensky, 1986] for an introduction to Lisp, and Kowalski [Kowalski, 1974][Kowalski, 1979] and Bowen [Bowen, 1979] for PROLOG.

In some sense, there are two user interfaces that are supported. The simpler one is the programmatic user interface: that is, the functions that are supplied for a programmer to gain access to Rhet. From such an interface, a user can expect to be able to do proofs, add facts and axioms, but not, perhaps, reorder axioms or debug much in the way of Rhet problems. The user who utilizes Rhet's more screen oriented tools for editing axioms, doing queries and making assertions will garner a definite benefit for debugging his system. It is possible, of course, to have the best of both worlds: namely, use the screen oriented utilities to get up that part of your system you expect Rhet to do, and then tie your own software in via the more primitive interface. This choice, is left to the user, however.

The following sections define the Rhet language as supported by the interface, the programmatic interface, and describes the enhanced screen-oriented interface[3]. Notes in the margin point out the significant departures from the original HORNE system. A certain minimal amount of familiarity with Lisp, and Prolog, is assumed[4]. For further internal details about Rhet, see [Miller, 1989].

A small initial note on notation used herein: Throughout we use "()"s to denote Lisp lists, in a manner familiar to Lisp users. We use "[]"s to denote Rhet objects, such as axioms, expressions, or facts. In fact, the Rhet program expects these symbols in it's input as well, to disambiguate between Rhet and Lisp objects, as described in detail in section 7.1. A rhet object that appears alone, e.g.

(1.1) `[[F ?x] <foo [P ?x] [Q ?x]]`

is considered to be an assertion, that is, we are adding the above statement to the KB. Briefly, in the above statement, a horn clause or an axiom, the clause to the left of "<foo" is the left hand side *form* of the axiom,

[3]For the most part, the screen oriented interface functionally builds on the existing facilities, such as ZMACS, of the Explorer™ and Symbolics™ environments. It is not expected to be particularly portable to non-Lispms, unlike the rest of the system. This is not to say that the rest of the system will not require porting, but for the most part, all machine dependent code is isolated.

[4]And to a small extent Lisp machines.

usually abbreviated as the *LHS*, and the clauses to the right of "<foo" are considered the right hand side forms, usually abbreviated as the *RHS*. The token "<foo" itself is referred to as the index of the axiom. The capitol letters are constants, and in this case are all predicates (they occupy the first position in a clause) and the symbols beginning with a question mark are all variables.

The above is distinguished from, for example,

(1.2) ((F ?x) <foo (P ?x) (Q ?x))

which is not an axiom at all, but a lisp list with four elements, a list, an atom, and two more lists. Note that even in this example, however, the symbol ?x is still a Rhet variable, even though the capitol letters are all lisp atoms.

1.2 Acknowledgments

Special acknowledgment is given to Michael McInerny, who has invested a considerable amount of RA and programming time into the user interface, type subsystem, and user contexts; Stephane Guez for his work on structured types; Jun Tarui for his work on TMS and function typing; Nat Martin for his RA work on the parser; Steven Feist, for his RA work on the type subsystem; Jay Weber for his helpful comments and coding assistance; and everyone else who gave us suggestions on the design, those who suffered thru HORNE and are giving their experiences, and even those who built HORNE in the first place...

1.3 Help Us!

We want to know about problems you have using Rhet, inconsistancies with the manual, or suggestions on how any part of Rhet might be improved (*e.g.* to make it easier to use, the manual more clear, the

index more functional, *etc.*). Send mail to one of the rhet discussion lists: Rhet@cs.rochester.edu or Bug-Rhet@cs.rochester.edu depending on whether you are asking a question, or proposing an enhancement (the former list is appropriate), or reporting a bug or inconsistancy. You may also contact the author directly using the address given on the title page of this document.

Part I

Tutorial Introduction to Rhet

Chapter 2

The Basic Reasoning Modes

This is a brief introduction to the major reasoning modes and facilities provided by the Rhet reasoning system. In this section, we will first discuss the basic reasoning modes, and then outline the specialized reasoning systems embedded in Rhet.

There are three basic reasoning modes. The first two correspond to the antecedent and consequent theorem mechanisms of PLANNER, and are called forward chaining and backward chaining, respectively. The third is most closely related to reasoning with constraints, and is called constraint posting.

Independent of the mode of reasoning, all facts are in the form of horn clauses, which can be viewed as logical implications with a single consequent. Thus `[P < Q]` read as "if Q then P," is a horn clause, as is `[P <]` which simply asserts P, as is `[P < Q R]` which should be read as "if Q and R, then P." RHET also supports a limited form of reasoning about negation, and thus `[[Not P] <]` is allowed, as is `[P < [Not Q] R]`. The interpretation of these formulas will be described in the next section. The following is not a horn clause, because there are two consequences: `[P Q < R]`. Note that, in more general systems of this type, this would be read as "if R, then P or Q."

HORNE did not support negation.

A horn clause may contain globally scoped, universally quantified variables which are indicated by a prefix of "?". Thus `[[P ?x] < [Q ?x]]` is a horn clause that is read as "for any x, if Q of x holds, then P of x holds." Finally, whenever the process of matching two formulas is discussed, we are referring to the full unification algorithm found in resolution theorem-proving systems.

2.1 Backwards Chaining

This mode provides a PROLOG [Bowen, 1979]-like theorem prover. It searches a horn clause that could prove the given goal, and attempts to prove the antecedents of the horn clause. It uses a depth-first, backtracking search. For the reader not familiar with such systems, see [Kowalski, 1979].

2.1.1 Simple Reasoning Mode

Contrast to HORNE

Rhet supports three distinct reasoning modes when using backwards chaining. It's simpler reasoning mode uses the usual PROLOG algorithm for a proof using the closed world assumption. In this mode, Rhet assumes failure to prove is enough for a proof of the negation. As an example, consider the following axioms:
All fish live in the sea.

(2.1) `[[LIVE-IN-SEA ?x] < [FISH ?x]]`

All Cod are fish.

(2.2) `[[FISH ?x] < [COD ?x]]`

All Mackerel are fish.

(2.3) `[[FISH ?x] < [MACKEREL ?x]]`

Whales live in the sea.

(2.4) `[[LIVE-IN-SEA ?y] < [WHALE ?y]]`

Homer is a Cod.

(2.5) `[[COD HOMER] <]`

Willie is a Whale.

(2.6) `[[WHALE WILLIE] <]`

Given these axioms, we can prove Willie lives in the sea as follows, using a straightforward backtracking search. We have the goal:

(2.7) `[LIVE-IN-SEA WILLIE]`

Rule 2.1 appears applicable: Unifying 2.1 with 2.7 we get

(2.8) `[[LIVE-IN-SEA WILLIE] < [FISH WILLIE]]`

So we have a new subgoal:

(2.9) `[FISH WILLIE]`

Rule 2.2 applies, giving

(2.10) [[FISH WILLIE] < [COD WILLIE]]

so we have a new subgoal

(2.11) [COD WILLIE]

⌢ No rule applies, try 2.9 again.
Rule 2.3 applies, giving

(2.12) [[FISH WILLIE] < [MACKEREL WILLIE]]

So we have a new subgoal

(2.13) [MACKEREL WILLIE]

⌢ No rule applies, try 2.9 again, no more ways to prove 2.9
⌢ No rule applies, try 2.7 again
Rule 2.4 applies giving

(2.14) [[LIVE-IN-SEA WILLIE] < [WHALE WILLIE]]

So we have a new subgoal

(2.15) [WHALE WILLIE]

Rule 2.6 asserts 2.15 as a fact

⌣ Goal 2.15 is Proved.

⌣ Goal 2.7 is Proved.

Further, assume we are attempting to prove the form `[Not [FISH WILLIE]]`. A typical horn clause prover does not deal with negation. Our system on the other hand would deal with this proof as follows:

First, look for rules that unify directly with the goal. We fail because we have no axioms whose LHS include the token "Not".

Next, attempt to prove the Cdr of the Not form. If we fail, then we have proved the negation[1]. In this case our proof is the same as our proof above of 2.9. Since this fails, our Not goal succeeds[2].

Note that in general we allow `[Not]` forms to be asserted both as facts and inference rules. That is, both of the following are valid clauses in Rhet:

(2.16) `[[Not [WHALE ?X]] < [FISH ?X]]`

(2.17) `[[Not [HUMAN FIDO]] <]`

where the first, 2.16 says that anything that is a fish is not a whale[3]. Note that having such a rule does not imply it gets used. When Rhet is doing simple reasoning, Not forms of rules will *only* be invoked when it is *explicitly* attempting to prove a `[Not]` form. If it cannot so prove it, it returns failure. Assertion 2.17 does what one would expect: it explicitly asserts that `[HUMAN FIDO]` should fail. Again, if this contradicts some other rule, such as `[[HUMAN ?X] < [NICE ?X]]` given `[NICE FIDO]` we would be able to prove `[HUMAN FIDO]`

[1] Naturally this only works if our axioms embody the "closed world assumption".

[2] In fact, RHET will return :Unknown for this proof since it could neither prove nor disprove it's goal. The user is free to consider :Unknown to be success or failure as they desire.

[3] Or more specifically, if we want to prove that something is not a whale, we can do so by proving it is a fish.

if that were our goal, and [Not [HUMAN FIDO]] if that were instead our goal. In *simple* mode, the system would not detect any contradiction.

Of course, normally, if we are using the closed world assumption, we would have no reason to use these [Not] forms on the LHS of a horn clause. We would still be free to use them on the RHS, however. That is, we would make the Not form a goal, as in the example above.

2.1.2 Complete Reasoning Mode

To make such contradiction detection possible, Rhet also has a mode wherein it both attempts to prove and disprove a goal. Normally this would be used for detecting contradictions, or for question answering where the closed world assumption doesn't hold. When we suspend this assumption, we need to be able to give rules on how to prove that something is false, or be able to explicitly assert that something is false. The [Not] forms outlined in the previous section serve this need well.

The system, now, can return four different results in complete reasoning mode. It can succeed, that is, prove the goal. It can disprove the goal. It can fail to either prove or disprove the goal, and it can detect a contradiction. It's mechanism for doing this is outlined as follows:

Given a goal (or subgoal) G:

- The system tries to prove G as in simple mode.
- The system tries to prove the complement of G as in simple mode.

We now have 4 possible results for each of the 2 main proof types (*i.e.* attempting to prove and attempting to disprove). Our returned result will depend on each of them, as in table 2.1.

So for the example at the end of the last section, we would indeed derive the contradiction. However, such things are also affected by the *mode* Rhet is in. Section 10.7 has more details, but by default, Rhet,

Not Form⇒ Form ⇓	Prove Succeeds	Prove Fails
Prove Succeeds	Contradiction	True
Prove Fails	False	Unknown

Table 2.1: Complete Proof Truth Table

during a complete proof, will prune any subtree that has a contradiction, although such contradictions will be reported[4]. For example, if we wish to prove

```
[P A]
```

and we have the rules

(2.18) `[[P ?x] < [G ?x]]`

(2.19) `[[P ?x] < [Not [Q ?x]]]`

(2.20) `[[G ?x] < [R ?x]]`

(2.21) `[[R A] <]`

(2.22) `[[Not G A] <]`

(2.23) `[[Not Q A] <]`

[4] At least, by default. See 10.7.

Since order is significant[5], we first get the subgoal `[G A]` which we can prove if we can prove `[R A]`. Since we can prove `[R A]`, and not `[Not R A]` this succeeds. We now try to prove `[Not G A]` and also succeed, so this branch is marked as inconsistent. Thus our first rule fails, and we attempt to prove `[P A]` by proving `[Not Q A]`. Since this succeeds, and we have no proof for `[Q A]` we succeed in proving `[P A]`. Note that had we not used the complete reasoning mode, we would have succeeded using our first rule, and never detected the contradiction.

2.1.3 Question Answering Mode / Default Reasoning Mode

Not to be confused with the specialized reasoning system about defaults (default reasoning vs. "default" reasoning mode). A third reasoning mode is available, and is in fact the default. In this mode, only the top level goal is proven using the complete reasoning mode. All subgoals are proved using the simple reasoning mode. Since the complete reasoning mode does include the possibility of being exponentially harder[6], this mode is recommended as a starting point. There are certain conditions where this mode will not work, in particular, whenever default reasoning is engaged, and Rhet is reasoning about default axioms (see section 3.4). Since Rhet normally takes pains to cache its subgoals and proofs, complete reasoning mode usually will not be a significant performance problem, relative to the additional information it can supply.

Goal caching is not currently operational.

2.2 Forward Chaining

The "rules" for forward chaining are quantified horn clauses augmented with a trigger. Such a rule is applied whenever a fact is added that matches (*i.e.*, unifies with) the trigger. In such a case, the reasoner attempts to prove the antecedents of the rule and, if it is successful, asserts the consequence. In general, each of the

[5]Actually it is undefined which we get first, but for illustrative purposes...

[6]Particularly if goal caching is disabled.

antecedents is attempted by simple data base lookup only. In other words, the backwards chaining reasoner is not invoked to prove an antecedent. There is an option, however, to invoke the backwards reasoning if desired[7].

In addition, assertions made via the forward chainer are 'tagged' with the justification, or support for the assertion. This is to allow automatic retraction of forward chained assertions if the support for the assertion has been lost.

For example, consider maintaining the simple transitive relation < (less than) using forward chaining. The axiom we want to use to ensure the complete KB is

$$\forall x, y, z \; LT(x, y) \wedge LT(y, z) \Rightarrow LT(x, z) \tag{2.1}$$

To implement this using forward chaining rules, we have the following[8]:

(2.24)
```
[[LT ?x ?z] < [LT ?x ?y] [LT ?y ?z]
              :forward [LT ?x ?y]]
```

(2.25)
```
[[LT ?x ?z] < [LT ?y ?z] [LT ?x ?y]
              :forward [LT ?y ?z]]
```

[7]Note that such an invocation should probably use *complete* proofs in order to prevent contradictions from being introduced.

[8]Note that forward chaining rules are notated like regular horn clauses, but have the additional keyword :Forward on the RHS, and the clauses that follow it are the triggers for the rule (Note that while multiple triggers may be specified, Rhet will store FC rules with only one trigger each. Thus one can have the non-intuitive result of adding a FC axiom with a trigger of :All and end up with several FC axioms being asserted and listed via the FC axiom listing functions.). If no clause follows, or the keyword :All is used, then all of the RHS forms are used as triggers.

Trigger	Rule
[LT B C]	triggers rules 2.24 and 2.25, but nothing can be proved
[LT A B]	triggers 2.24 ?x ← A, ?y ← B proves [LT A B] ⌣ proves [LT B ?z], ?z ← C adds [LT A C] with support 2.24 [LT B C] and [LT A B] triggers 2.24 ?x ← A, ?y ← C proves [LT A C] ⌣ fails on [LT C ?z] ⌢ triggers 2.25 ?y ← A, ?z ← C proves [LT A C] ⌣ fails on [LT ?x A] ⌢ triggers 2.25 ?y ← A, ?z ← B proves [LT A B] ⌣ fails on [LT ?x A] ⌢

Table 2.2: Simple FC Trace

Consider the additions made in table 2.2. As one can see, the rules apply recursively on inferred additions, and the search space generated by the forward chaining rules is completely searched. The forward chainer never adds the same fact twice[9].

Now if, for some reason, we were to retract `[LT B C]`, we could expect `[LT A C]` to be retracted as well since the former supports the latter. In this example, `[LT A C]` had no other parallel support, but in general we would want to reattempt to prove `[LT A C]`, using the FC axioms, in a mode similar to backward chaining.

Current Status:
Not operational.

2.3 Constraint Posting

The last facility allows proofs of goals to be delayed for certain predicates until more is known about the arguments to the predicate. In particular, it allows one to delay proving a formula until one of its variables is bound.

This is best illustrated by example. Assume we want to define a predicate of two arguments, ?x and ?y, that is true iff ?x and ?y are bound to different terms. The most common way to implement this in PROLOG systems is to use negation by failure on a naive EQ predicate, which is simply defined by

(2.26) `[EQ ?x ?x]`

Thus EQ forces two terms to unify, and fails if they cannot. Using this, they define

(2.27) `[[NotEQ ?x ?y] < [Unless [EQ ?x ?y]]]`

[9] This has an implication: the justifications stored with a particular fact derived via FC is not "complete", that is, there may have been other FC rules that could have added a particular fact. Rhet will deal with this case only during retraction, where it will then attempt to find some other rule that can support the fact being retracted. At the moment, this functionality of finding alternate support may not be fully operational, however.

where Unless is negation by failure. This formulation gives undesirable results when one of its terms is unbound. In particular, it binds a variable argument to make the terms equal. Thus with the axioms

(2.28) `[[P ?x ?y] < [NotEQ ?x ?y] [R ?y]]`

(2.29) `[R B]`

we could not prove `[P A ?y]` for the predicate `[NotEQ A ?y]` would fail since `[EQ A ?y]` succeeds by binding ?y to A.

To avoid this, we could define NotEQ so that it only fails when both arguments are bound. But this would allow incorrect proofs as the variable could later be bound violating the distinctness condition. What is needed is a facility to delay the evaluation of `[NotEQ ?x ?y]` until both arguments are bound. We do this by a mechanism called posting.

If a literal is POSTED and contains no variables, it is treated as a usual literal. The proof succeeds or fails and the posting has no effect. If the literal does contain a variable, the evaluation of that literal is delayed until the variable is bound. Thus we define a new predicate Distinct by

(2.30) `[[Distinct ?x ?y] < [Post [NotEQ ?x ?y]]]`

Now, using a modified axiom 2.28, namely,

(2.31) `[[P ?x ?y] < [Distinct ?x ?y] [R ?y]]`

and the modified definition of `[NotEQ]` as in axioms 2.32—2.34, *i.e.*, `[NotEQ ?x ?y]` is true if either ?x or ?y is not fully grounded (*i.e.*, it is a term containing a variable), or if the two grounded terms cannot be proven to be equal:

(2.32) `[[NotEQ ?x ?y] < [Unless [Ground ?x]]]`

(2.33) `[[NotEQ ?x ?y] < [Unless [Ground ?y]]]`

(2.34) `[[NotEQ ?x ?y] < [Unless [EQ? ?x ?y]]]`

Given clauses 2.29 through 2.34, we can prove `[P A ?y]`, resulting in ?y being bound to B as follows:

Goal: `[P A ?y]`

Subgoals: `[Distinct A ?y]` `[R ?y]`

`[Distinct A ?y]` is proven using 2.30, but the subgoal `[NotEQ A ?y]` is not evaluated in the normal manner since ?y is unbound. Instead, the call succeeds and ?y is annotated to be NotEQ from A.

`[R ?y]` succeeds from axiom 2.29 if ?y can be bound to B. The unifier checks `[NotEQ A B]`, which succeeds, allowing ?y to be bound.

Thus the goal proved is `[P A B]`. Note that `[Distinct]`, `[Ground]`, and `[NotEQ?]` are built-in predicates in Rhet and are defined using these mechanisms.

Let us consider this mechanism in a bit more detail. After a literal Q has been POSTed, its variables are annotated using a form such as `[Any ?x [Q ?x]]` which is a term that will unify with any term such that Q holds for that term. Thus `[Any ?x [Q ?x]]` unifies with A only if we can prove `[Q A]`.

If there are multiple variables in a posting, each variable is annotated separately, and the constraints on each are checked as each is bound. For example, the trace of the proof of `[P ?x ?y]` given axioms 2.29—2.34 is as follows:

Goal: `[P ?x ?y]`

Rule 2.31 applies, giving

`[[P ?x ?y] < [Distinct ?x ?y] [R ?y]]`

Subgoal `[Distinct ?x ?y]`

Rule 2.30 applies, giving

[[Distinct ?x ?y] < [Post [NotEQ ?x ?y]]]

Subgoal [POST [NotEQ ?x ?y]]
succeeds constraining ?x ← [any ?x1 [NotEQ ?x1 ?y1]]
?y ← [any ?y1 [NotEQ ?x1 ?y1]]

Proved: DISTINCT [any ?x1 [NotEQ ?x1 ?y1]]
[any ?y1 [NotEQ ?x1 ?y1]]

Subgoal [R [any ?y1 [NotEQ ?x1 ?y1]]]

Rule 2.29 applies—

[R B] if we can unify [any ?y1 [NotEQ ?x1 ?y1]] with B

[We try subproofs of [NotEQ ?x1 B], which succeeds]

Proved: [P [any ?x1 [NotEQ ?x1 B]] B]

Thus constrained variables may appear in answers. Users may explicitly construct their own constrained variables in queries and assertions as well, if they wish.

Two constrained variables may unify together as long as the combined constraints are provably consistent in a strong sense, *i.e.*, there exists at least one proof of the combined constraints. For example, if we had the following data base:

(2.35) [PA A]

(2.36) [PB B]

(2.37) [PB A]

(2.38) [T [any ?x [PA ?x]]]

We could prove the goal [T [any ?y [PB ?y]]] by unification with 2.38 as follows: [any ?y [PB ?y]] and [any ?x [PA ?x]] may unify to [any ?z [PB ?z] [PA ?z]] if there is an object such that [PB ?z] and [PA ?z]. A subproof of [PB ?z] [PA ?z] is found with ?z ← A. This binding is not used, however, since the desired answer could be something else. The result is [T [any ?z [PA ?z] [PB ?z]]].

If in a later part of a proof, ?z was unified against a constant k, a subproof of [PA k] [PB k] would be done before the unification succeeds.

Chapter 3

Built-In Specialized Reasoning Systems

There are two[1] built-in specialized reasoning systems provided with Rhet. These provide typing for terms and simple equality reasoning.

3.1 Types

All terms in Rhet may be assigned a type. If a term is not explicitly assigned a type, it is assumed to belong in *T-U, the universal type. Variables over a type are allowed, and a special syntax is provided. The variable ?x*Dog, for instance, signifies a variable ranging over all objects of type *Dog[2]. Constants and other ground terms can be asserted to be of a certain type using built-in predicates Itype, Dtype, and Utype. Thus (Itype 'Dog [A]) asserts that the constant [A] is of type *Dog, and further, that *Dog is it's 'immediate' type: it

[1]The TEMPOS system makes a third.

[2]In other words, it will only unify with objects that are provably in the class *Dog.

cannot be further specialized. Utype, on the other hand is 'unconstrained' type, and only implies that the object is a member of the type considered as a set. It may in fact be a member of a subset of the type too.

Types in Rhet are viewed as sets of objects, and all the normal set relationships between types can be described. Thus one type may be a subset (*i.e.*, subtype) of another, two types may intersect or be disjoint, and the non-null intersection of two types produces a type that is a subtype of the two original types. All this information is asserted using Lisp functions, or the fancy user interface.[3] For example, (Tsubtype 'Animal 'Dog) asserts that the type *Dog is a subset of the type *Animal (*i.e.*, all dogs are animals), (Tdisjoint 'Dog 'Cat) asserts that no object can be both a cat and a dog, (Tname-Intersect 'Fat-Cats 'Cats 'Fat-Animals) asserts that the set of *Fat-Cats consists of all cats that are also fat animals, and (Txsubtype 'Animal 'Males 'Females) asserts that Males Females is a partition of *Animal, *i.e.*, that every animal is either a male or a female, and that all males and females are animals.

All direct consequences of these facts are inferred when the type assertions are added. For example, if *A and *B are disjoint, and *A1 is asserted to be a subtype of *A, then it is inferred that *A1 and *B are disjoint. This is done by a constraint propagation subsystem.

The type reasoner acts during unification. A constant will match a variable of type *Tv only if the constant is of type *Tv (*i.e.*, the constant is asserted to be of type *Tv, or is of type *Tvs which is a subtype of *Tv). Two variables unify only if the intersection of their types is non-empty. The result is a variable ranging over the intersection of the two types. Thus, complex types may be constructed during a proof. If types *T1 and *T2 intersect, but no name for the intersection is asserted, then a complex type *(T1 T2), which is their intersection, is constructed when unifying ?x*T1 and ?y*T2.

The type calculus also allows type subtraction. For example, the type of all animals except dogs is *(Animal - Dog). In general, a type expression is of the form $[t_1 \ldots t_n - s_1 \ldots s_m]$, and is the class of objects in the intersection of types t_1 through t_n except for those objects in s_1 through s_m. For example, the type of all male animals except dogs and cats would be *(Animal Males - Dog Cat).

[3] In HORNE, type information was asserted as axioms. This is not supported in Rhet. All type information is entered directly to the type system by the functions described here.

This type reasoner provides a complete reasoning facility between simple types. For complex types, however, the reasoner may permit some intersections that may not be desired since they are empty. Note that this can be checked for at the end of a proof if desired. Any intersection of more than two types is guaranteed only to be pairwise non-empty. For example, if the complex type *T1 T2 T3 is constructed by unifying a variable of type *(T1 T2) with a variable of type *T3, then it must be the case that *(T1 T2), *(T1 T3), and *(T2 T3) are non-empty. However, there might be no object that is of type *(T1 T2 T3).

The assertions about the types may be incomplete. For example, two types may be introduced where it is not asserted, or is inferable, that the types intersect or are disjoint. Rhet provides two modes of proof for dealing with these cases. In the strict ("Default") mode, two types intersect only if they are known to intersect. In the easy-going ("Assumption") mode, two types will intersect unless they are known to be disjoint. Easy-going mode is more expensive, but can be useful in many applications, although it may provide conclusions that on closer inspection are not useful since they contain a variable ranging over the empty set.

Lisp objects are also given types by Rhet. Lists are of type T-List, atoms are of type T-Atom and both are subtypes of T-Lisp, which is not a subtype of T-U. T-U and T-Lisp are defined to be subtypes of an all-encompassing type: T-Anything. The type T-Nil is defined to be disjoint from every other Rhet type.

As an example, the simple fish data base above could be restated in the typed prover as follows:

(3.1) (TSUBTYPE 'FISH 'COD 'MACKEREL)

(3.2) (TDISJOINT 'FISH 'WHALE)

(3.3) (ITYPE 'COD [HOMER])

(3.4) (ITYPE 'WHALE [WILLIE])

(3.5) [[LIVE-IN-SEA ?x*FISH] <]

(3.6) [[LIVE-IN-SEA ?y*WHALE] <]

Although this took one more assertion (if you count the two Lisp calls), it also encodes more information (*e.g.*, whales and fish are disjoint). The proof that WILLIE lives in the sea is much shorter in the typed system. It is completed using only two unifications.

Goal: [LIVE-IN-SEA WILLIE]

unifying with 3.5 fails as WILLIE is not a fish;

unifying with 3.6 succeeds, ?y ← WILLIE.

Thus the goal is proved.

If we add the following axioms, we can demonstrate more complicated type reasoning. Let us assume that all animals are either fish or mammals.

(3.7) (TXSUBTYPE 'ANIMAL 'FISH 'MAMMAL)

This asserts that both FISH and MAMMAL are subtypes of ANIMAL and that they are disjoint. Note that since COD and MACKEREL are subtypes of FISH, these will also now be disjoint from MAMMALS.

(3.8) (TSUBTYPE 'MAMMAL 'WHALE)

This asserts that WHALE is a subtype of MAMMAL, and hence WHALE is disjoint from FISH.

(3.9) (TSUBTYPE 'THINGS-THAT-SWIM 'WHALE)

(3.10) (TSUBTYPE 'THINGS-THAT-SWIM 'FISH)

Note that in asserting that WHALE is a subtype of THINGS-THAT-SWIM, the system then knows that MAMMAL and THINGS-THAT-SWIM intersect.

(3.11) `[[BEAR-LIVE-YOUNG ?m*MAMMAL] <]`

(3.12) `[[SWIMS-WELL ?t*THINGS-THAT-SWIM] <]`

Now if we try to find something that bears live young and swims well, i.e., find ?x such that `[BEAR-LIVE-YOUNG ?x]` and `[SWIMS-WELL ?x]`, we succeed by unifying the first subgoal to 3.11, causing ?x ← ?m*MAMMAL, and the second subgoal to 3.12, causing ?m*MAMMAL and ?t*THINGS-THAT-SWIM to be unified, resulting in a complex variable ?y*(MAMMAL THINGS-THAT-SWIM). Thus the answer is: all things that are both of type MAMMAL and THINGS-THAT-SWIM. If we add

(3.13) `[[LARGE ?w*WHALE] <]`

and query for something that bears live young, swims well, and is large, we will end up unifying ?y*(MAMMAL THINGS-THAT-SWIM) with ?w*WHALE. The result of this is simply ?w*WHALE, since *Whale is a subtype of both *Mammal and *Things-That-Swim.

Constrained variables may be typed in the obvious manner. For example `[Any ?x*MAMMAL [SWIMS-WELL ?x]]` is a term that will unify with any term t such that t is of type *Mammal, and `[SWIMS-WELL t]` is provable. It is interesting to note that the constrained variable system could be used to implement a typed system directly, where a variable ?x*MAMMAL would be replaced by `[Any ?x [TYPE ?x MAMMAL]]`. The semantics of the two notations are identical. Types are so common, however, that the special notation for variables is maintained and types are optimized in the implementation.

Unification between a typed constrained variable and a typed variable works as you would expect. That is, unifying ?x*Mammal with `[Any ?y*ANIMAL [SWIMS-WELL ?y*ANIMAL]]` succeeds with the result `[Any ?z*MAMMAL [SWIMS-WELL ?x*MAMMAL]]`. Unifying ?x*ANIMAL with `[Any ?y*MAMMAL [SWIMS-WELL ?y]]` succeeds simply and ?x*Animal is bound to the constrained variable.

Unifying a constrained variable with a term that itself contains variables may introduce new constrained variables. For example, if we are given the fact `[P [f A]]`, then unifying `[Any ?x [P ?x]]` with `[f ?w]` will

Term 1	Term 2	Most General Unifier
[Any ?x*MAMMAL [SWIMS-WELL ?x]]	[WILLIE]	[WILLIE]
[Any ?x*MAMMAL [SWIMS-WELL ?x]]	?a*ANIMAL	[Any ?x*MAMMAL [SWIMS-WELL ?x]]
[Any ?x*MAMMAL [SWIMS-WELL ?x]	?w*WHALE	[Any ?z*WHALE [SWIMS-WELL ?z]]
[Any ?x [SWIMS-WELL ?x]]	[SPOUSE ?a]	[SPOUSE [Any ?z [SWIMS-WELL [SPOUSE ?z]]]] assuming that the query [SWIMS-WELL [SPOUSE ?a]] succeeds.
[Any ?x [SWIMS-WELL ?x]]	[Any ?y [BEAR-LIVE-YOUNG ?y]]	[Any ?z [SWIMS-WELL ?z] [BEAR-LIVE-YOUNG ?z]] assuming that the queries [SWIMS-WELL ?z] and [BEAR-LIVE-YOUNG ?z] succeed.

Table 3.1: Unification with Constrained Variables

produce the term [f [Any ?z [P [f ?z]]]]. This is the correct result since the constrained variable ?x will unify with any term such that [P ?x] is provable. Since [P [f ?z]] is provable (because of the fact [P [f A]]), the terms unify. Note that the variable ?w is not bound to [A], however, since there may be other terms for which [P [f ?z]] holds as well. Thus [P [f A]] might not be the most general unifier.

These examples are summarized in Table 3.1[4].

[4]Note that some of the entries, more specifically the last two, will depend on the proof mode selected, see section 10

3.1.1 Typing Functions

Because of the additional complexities involved, a special system is provided for typing functions. This is needed for reasoning about function terms that contain variables. If the only functions used in the system are always fully grounded, the standard type system can be used directly.

For a given function, one can specify the type of the result of the function, plus the types on the arguments of the function. Any function term whose arguments violate these typing restrictions will signal an error[5]. Thus if we define the function SPOUSE to map from PERSON to PERSON, the term [SPOUSE WILLIE] will cause a warning, since WILLIE is a WHALE and thus cannot be a PERSON. This function could be defined as follows: (Declare-FN-Type 'SPOUSE '(PERSON PERSON)), *i.e.*, the function SPOUSE takes one argument of type PERSON, and produces objects of type PERSON.

Of course, one might like to do better than this, and define SPOUSE to be of type MALE when the argument is FEMALE, and FEMALE when the argument is MALE. Such definitions are allowed in Rhet. In other words, (Declare-FN-Type 'SPOUSE '(FEMALE MALE)) is allowed in addition to the above, as is (Declare-FN-Type 'SPOUSE '(MALE FEMALE)). This will produce the appropriate results during unification. Thus if we unify [SPOUSE ?m*PERSON] with ?x*MALE, the result is [SPOUSE ?m*WOMAN], as desired.

To be useful, the multiple specification of a function's type must be consistent. For example, if one asserted (declare-fn-type 'f '(T-U PERSON)) and (declare-fn-type 'f '(MALE DOG)), *i.e.* that f maps everything to PERSONS, and MALES to DOGS, then the type of (F ?x*MALE) would be *(PERSON DOG), namely *Nil.

Function typing does not guarantee that functions fully cover their range type (*i.e.*, they are not necessarily "onto"). For example, given

(3.14) (declare-fn-type 'G '(T-U) 'ANIMAL)

[5] I use the convention presented in [Steele Jr., 1984], that the phrase "signal an error" means invoking the debugger or a warning, while "is an error" means that no valid Rhet program will do such a thing, and the implementation is free to do something completely unpredictable. Doing something that "is an error" could possibly cause problems much later that seem unrelated to the original problem.

the query

```
[EQ? [G ?x] ?w*WHALE]
```

will fail[6], since there is no guarantee that any terms of form `[G ?x]` are of type WHALE, even though all are of type ANIMAL. If there is a known instance of G of type WHALE, such as `[Add-EQ [G ABLE] WILLIE]`, the above proof will succeed, but with ?x constrained to `[Any ?x [EQ? [G ?x] ?w*WHALE]]`.

Note that when defining the type of a function, the first definition counts as the maximumal (most general) arguments the function will accept. That is, given:

(3.15) `(Declare-Fn-Type 'R-SPOUSE '(T-HUMAN T-HUMAN) '(MALE FEMALE) '(FEMALE MALE))`

where we know the intersection of MALE and T-HUMAN is MAN, and the intersection of FEMALE and T-HUMAN is WOMAN, the form `[R-SPOUSE ?X*MALE]` will generate a warning, and be interned (and print out) as `[R-SPOUSE ?X*MAN]`; the function type declaration has already asseted the most general type accepted by the R-SPOUSE function will be of type T-HUMAN, and MAN is the intersection of MALE and T-HUMAN.

More precisely, the semantics of function specification is as follows:

```
(declare-fn-type f (1-type-1 1-type-2 ...1-type-n+1)
                   (2-type-1 2-type-2 ...2-type-n+1)
                   ...
                   (m-type-1 m-type-2 ...m-type-n+1))
```

is that given f(X_1 X_2 ...X_n):

1. For each $i = 1, 2, \ldots, m$, if X_i is compatible with type j-type-i for each $i = 1, 2, \ldots, n$, then f(X_1 X_2 ...X_n) is of type j-type-n+1 (or a subtype). That is, f(X_1 X_2 ...X_n) is in the intersection of all such j-type-n+1's where the X_i's match. Thus semantically the order in which function specification clauses appear as the argument doesn't matter, except that the first clause must be the most general.

[6] Like many things, there is an option that can override this. See section Options.

2. For all $j = 1,2,\ldots,m$, if $(X_1 \ldots X_n)$ is provably disjoint from (j-type-1 ...j-type-n) (this happens iff at least for one i X_i is disjoint from j-type-i), then f$(X_1 \ldots X_n)$ is of type "Nil". That is, declaration is supposed to "exhaust" all the possible argument types. [7] A user can add ("type-T" "type-T" ..."result-type") as one clause to express that function-atom can be applied to any type or more typically a user can add one clause corresponding for the most general types of arguments and result.

Less formally, for a given function term, the type of the term is, or is a subtype of the declared result type whenever each argument is or is a subtype of the declared argument types. The inverse must also be true: If we know the type of a function term, then the arguments must be subtypes of all function type definitions of those arguments for all result supertypes of the known term type, and further must be subtypes of at least one declaration whose result type matches the term type.

For example, given the definition of spouse above, repeated here as a single definition: (declare-fn-type 'SPOUSE '(PERSON PERSON) '(MALE FEMALE) '(FEMALE MALE)), the type of (SPOUSE JILL), assuming (ITYPE 'FEMALE [JILL]), would be computed as the intersection of PERSON and MALE. If MALE were defined as a proper subtype of PERSON, the complex type *(PERSON MALE) would be simplified to MALE.

In inverse, given the term [SPOUSE ?x*T-U] which is know to be of type MALE, we would know that this result is a subtype of the declaration (FEMALE MALE), and thus the type of ?x would be specialized to FEMALE. Further, because the first declaration was (PERSON PERSON), which is the maximum allowed type of both the arguments and the results, the result would be specialized to the intersection of MALE and PERSON, or MAN, and the variable would be further specialized to the intersection of FEMALE and PERSON or WOMAN. The type of the variable could have had further effect, e.g. given appropriate[8] definitions of the CHILD type, and the above function type declaration, the form [SPOUSE ?X*CHILD] with result type MALE would be specialized into [SPOUSE ?X*GIRL] with result type BOY.

---

[7] If we handle type "Nil" in a reasonable way, we can more or less handle partially defined functions (or functions that should be applied to only certain types) this way.

[8] the intuitive ones will do.

According to this semantics what a user can express is somewhat like Horn clauses. If a user wants to express "f(X) is of type FEMALEPERSON if and *only if* X is of type MALE and type PERSON", s/he can do this using complex types in the function definition:

```
(Declare-FN-Type 'F '((MALE PERSON) FEMALEPERSON)
                    '((- MALE) (- FEMALEPERSON))
                    '((- PERSON) (- FEMALEPERSON)))
```

But if disjointedness of certain types has been asserted, the clauses like ones above may be unnecessary as in the example of 'spouse' function where the system can infer that `[spouse ?x*MALE]` is disjoint from type Male although (male (- male)) doesn't appear in the specification of 'spouse'. Thus a user more or less has the same kind of control over how s/he expresses specification as the kind of control s/he has over how to express facts.

3.2 Equality

The system offers full reasoning about equality for ground terms. Thus if you add:

(3.16) `(Add-EQ [A] [B])`

(3.17) `(Add-EQ [B] [C])`

(3.18) `(Assert-Axioms [[P A] <])`

you will be able to successfully prove the goal

(3.19) `[P B]`

as well as

(3.20) **[P C].**

Furthermore, given the assertion

(3.21) **(Assert-Axioms [[P [f A]] <])**

you will be able to successfully prove the goals

(3.22) **[P [f B]]**

and

(3.23) **[P [f C]].**

Adding

(3.24) **(Add-EQ [g A] [B])**

with the above assertions allows you to prove a potentially infinite class of goals, including

(3.25) **[P [g A]],**

(3.26) **[P [g B]],**

(3.27) **[P [g C]],**

(3.28) `[P [g [g A]]]`,

(3.29) `[P [g [g B]]]`,

etc., to arbitrary depths of nesting of the g function.

Nonground equality assertions cannot be added (*i.e.* no variables are allowed in assertions). A facility is offered for reasoning about equality for non-ground terms as follows. With a data base of equalities between grounded terms, one can prove an equality statement with variables in it and the variables will be bound appropriately. Backtracking is "supported" via the expedient of the POSTing mechanism.

(3.30) (Add-EQ [f B] [G])

(3.31) (Add-EQ [f A] [G])

and we try to prove

(3.32) `[EQ? [f ?x] G]`

?x will be replaced with the expression `[any ?x [EQ [f ?x] G]]` and the proof will succeed. In fact, ?x may be bound to `[B]` as we would expect, and could potentially be bound to `[A]` as well. This might be taken advantage of later in the proof where only one will suffice, and we avoid wasting the resources of having to actually pick a particular binding, and later backtrack.

Equality may be restricted in two ways. The Add-InEQ assertion can be made to make two expressions decidedly unequal. Or, the Dtype (distinguished type) assertion can be used to create an object of a particular type, and assert that it cannot be made equal to any other object that has been asserted to be a Dtype of that type. For example, to assert:

(3.33) (Dtype 'T-Bird [Tweety])

(3.34) (Dtype 'T-Bird [Francis])

(3.35) (Utype 'T-Bird [The-Robin-That-Keeps-Flying-Into-My-Window])

we can later discover, or add, that either Tweety or Frances is equal to The-Robin-That-Keeps-Flying-Into-My-Window, but both cannot be because they are distinguished subtypes of T-Bird.

3.3 Structured Types

3.3.1 Basic Structured Type Usage

Rhet supports a hierarchy of structured types akin to frame-based knowledge representations. This facility allows one to associate roles with a type, and it allows subtypes to inherit roles from their supertypes.

Formally, a role is a distinguished function associated with a type. In particular, the function is defined on all objects in the class named by the type. There are two ways to access the values of roles of a given object. The first is by using the appropriate function; the second is by using a special predicate named `[Role]`. For example, say for the type T-ACTION, we have an "actor" role. Then if A is an object of type action,

(3.36) `[f-actor A]`

is the actor of A, as is the value of ?x in

(3.37) `[ROLE A :R-ACTOR ?x].`

Either one of these constructs can be used to retrieve the actor role. The second method, using the [ROLE] predicate, is more general, as it allows the user to query role names as well as values. For example, we could find what role ?r an object X plays with A by the query

(3.38) `[ROLE ? ?r*T-Atom X].`

Certain types may have a set of role names that suffice to uniquely identify each object in that type. In other words, if two objects of that type agree on all their roles, then the objects must be identical. These we shall call functional types. For functional types, a function, called the *constructor function*, can be defined that maps the set of roles to the object that they identify. For example, if an event of type T-MELT is completely defined by the object melting (R-OBJECT), the time (R-TIME), and the location of melting (R-LOC), then we can define a function

(3.39) `[c-melt ?o*T-PHYS-OBJ ?t*T-TIME ?l*T-LOCATION]`

that generates the class of melting events.

Given this informal semantics, we can see that certain relations hold between constructor functions and role functions. In particular, if M is any melting event as defined above, then we know that

(3.40) M =`[c-melt [f-object M] [f-time M] [f-loc M]]`

even if we do not know the actual values of the three roles of M.

The Rhet system automatically generates the above function definitions and supports the required equality reasoning between objects, constructor functions, role functions, and the [ROLE] predicate.

Structured types are declared to the system using two commands, introduced here by example. Full details can be found in section 7.4 of this document. The command:

(3.41) (Define-Subtype 'T-ACTION 'T-EVENT :roles '((R-ACTOR T-ANIM)))

defines T-ACTION to be a subtype of T-EVENT with the role R-ACTOR defined. All values of R-ACTOR are of type T-ANIM[9]. In addition, T-ACTION will inherit any roles defined with the type T-EVENT.

The [ROLE] predicate is axiomatized such that any object O which is asserted to be the R-ACTOR of some action A will be equal to [f-actor A]. Thus if we add [Add-Role A :R-ACTOR O] then [Add-EQ [f-actor A] O] will automatically be asserted as well.

On the other hand, the command:

(Define-Functional-Subtype 'T-ACTION 'T-EVENT :Roles '((R-ACTOR T-ANIM)))

would do all of the above, and in addition defines a constructor function C-ACTION that takes an object of type T-ANIM and produces an object of type T-ACTION.

The system is set up so that any instance of type T-ACTION will be equal to its appropriate constructor function. Thus, if we now add (ITYPE 'T-ACTION [A]) the assertion [Add-EQ A [c-action [f-actor A]]] would be asserted as well.

With the equality reasoning abilities of Rhet, the system can now integrate all role values as they are asserted later and the appropriate conclusions regarding the equality of objects can be derived. Thus, if we add [Add-EQ [f-actor A] O] and [Add-Role A :R-ACTOR O'] then [Add-EQ O O'] will be concluded. Furthermore, the equalities [EQ? A [c-action O]] [EQ? [c-action O] [c-action O']] can be derived as needed during any proof.

Roles are automatically inherited from supertypes at the time the structured type is defined. These inherited roles will appear in constructor functions following the role values that were explicitly defined with the type. An inherited role may be redefined lower in the hierarchy only if the type restriction on the new role definition is a subtype of the original role definition. For example, assuming T-ACTION was a regular (non-functional) subtype of T-EVENT as defined above, if we use

[9] In particular, a function f-actor is defined that maps an object of type T-ACTION to an object of type T-ANIM.

(3.42) (Define-Functional-Subtype 'T-OBJ-ACTION 'T-ACTION :Roles '((R-OBJ T-PHYS-OBJ)))

a constructor function of the form

(3.43) [c-obj-action ?obj*T-PHYS-OBJ ?a*T-ANIM]

would be defined. On the other hand, the definition

(3.44) (define-functional-subtype 'T-SING 'T-ACTION :Roles '((R-ACTOR T-PERSON)))

would be allowed only if T-PERSON were a subtype of T-ANIM. If this were so, a constructor function of the form

(3.45) [c-sing ?a*T-PERSON]

would be defined.

3.3.2 Advanced Structured Type Usage

Some other supported functionality of the structured types:

Rhet will allow the definition of a type as a conjunction of other types. The new type will have all of the roles and type restrictions of the conjoined types. Should a role be defined on more than one of the conjoined types, the conjunction will be restricted to the most specific type. In addition, Rhet will allow conjoining roles that have different names among the conjoined types.

For example,

(3.46) (Define-Subtype 'A :Roles '((R-Actor Person) (R-Object Anything)))

and

(3.47) (Define-Subtype 'B :Roles '((R-Agent Person) (R-Object Phys-Obj)))

followed by

(3.48) (Define-Conjunction 'AB '(A B) :Roles '((R-Agent R-Actor)))

would define a new type AB with roles R-Agent of type Person, and R-Object of type Phys-Obj.

Type or instance classification is supported by Rhet. This would be used to determine where some object should fit into the taxonomy. This would allow us to get from the roles of an instance to it's likely type if we don't know. To do this, Rhet supports the Classify and Subsumes predicates.

Finally, Rhet supports a generalized predicate restriction between roles of a type. We may pass a keyword parameter to Define-Subtype to define the restrictions as predicates which are asserted. Equality constraints are asserted whenever a type instance is created or changed (as it is not allowed to assert equalities with variables). For example,

```
(Define-Subtype 'Grandmother 'Female-Person
                :Roles '((R-Child Person) (R-Grandchild Person))
                :Constraints '([EQ? [F-Grandchild ?self] [F-Child [F-Child ?self]]]))
```

says that for any object of the type Grandmother, the grandchild role of this object is equal to the child of the child of this object. Note the use of the variable ?self; this is bound to the new instance of this type when it is being instantiated.

3.4 Defaults

Rhet supports a limited amount of default reasoning, and truth maintenance. This section will be greatly expanded when it becomes clear just how much it really will support, as will the related axioms. The state of the current thought: Facts can be added to the system with a justification. While this justification may not be human readable, there should be interface functions making it so, and allowing programmatic (*i.e.* outside of Rhet) generation of justifications. Whenever Rhet does a proof, it will have available (unspecified how) the justifications for the proof. The user would be free to assert whatever was just proved including the justification Rhet provides.

Forward chaining would always add justifications to forward chained assertions.

What is a justification? It is the support for adding a fact (to a context). That is, given [A <], and [B < A], we could add [B <] with the justification or support of the first two axioms. The intent is that the TMS system would retract any fact whose support has been compromised. In this example, were we later to retract [A <], even in a subcontext (*q.v.*), in that context [B <] would also be retracted [10].

Certain axioms or facts may be marked as *default*. Default rules and facts are "visible" to Rhet only if default reasoning is enabled[11]. Even with defaults enabled, non-default rules and facts have "priority" over the default ones. That is, let us say the reasoner, using complete proof mode proves some form using only non-default forms and facts, and also disproves it, but must invoke a default rule to do so. This disproof would be considered "invisible" rather than a contradiction.

Another facet of this subsystem is its ability to deal with default roles in a structured type. [12] Normally, equality cannot be retracted, that is, once an equality assertion has been made in a context, in that context

[10] The retraction need not occur in the context that any of our supports or results were added into, but they must all be accessible from this context.

[11] This is a pretty primitive mechanism, meant as a stopgap for better things. It would allow a user to disable default reasoning, do a proof, and only if he could not do the proof enable it. Since such reasoning might be costly, it is possibly more efficient to do this.

[12] It is possible, though equality, to have several possible default rolevalues

and all subcontexts the equality is there to stay. A special KB keeps track of the values of roles for structured types, should no other concrete one be defined[13]. In this case, the default would be used.

Another task of TMS is to keep default rules consistent (or at least, as consistent as possible). For inference rules, only a certain class of rules is allowed to be marked as default. Specifically, only rules without a RHS, *e.g.*

Not currently implemented

(3.49) `[[CAN-FLY ?x*BIRD] < :default]`

which states that anything that matches the type-restricted variable CAN-FLY. Now let us say that we add the rule:

(3.50) `[[NOT [CAN-FLY ?x*PENGUIN]] < :default]`

where the type subsystem is already cognizant that a penguin is a subclass of bird. In this case, TMS will update the first rule to

(3.51) `[[CAN-FLY ?x*BIRD-PENGUIN] < :default]`

which expresses that any bird can fly unless it is a penguin. In so doing, the support for any facts added with the original rule is cut (since it was retracted), and TMS will attempt to reprove them to maintain consistency.

[13] Adding that the mother role of object A is equal to the mother role of object B is not considered to be concrete.

Chapter 4

Contexts

Rhet greatly expands the existing HORNE reasoning system with its concept of *Contexts*. A context is a space that axioms and facts can apply to. Rhet tracks contexts as a hierarchy, so the root context, "T", which is also the initial default context, contains facts and axioms that all other contexts inherit by default. Child contexts can be created and destroyed arbitrarily, though since contexts are a tree structure, destroying a child implicitly destroys all of its children.

There are two basic user interfaces to contexts that Rhet supports. One is for belief spaces, and one is for user contexts, including assumptions. The model we use for beliefs is that all models of the form XB inherit from MB, and we form a sort of a tree, as in figure 4.1.

The system can represent the beliefs of other agents (using the letters A-L, N-R, T-Z), but these are still relative to SB. In practice, the leading SB is dropped, so MB⇒SBMB in the above, while SB⇒SB. Notice that the letters "S" and "M" have special meanings. The system (or "speaker") is represented by the letter "S", while mutual belief operators end in "MB". It is traditional for the the other agent (if there's only one other) to be "H", for "hearer".

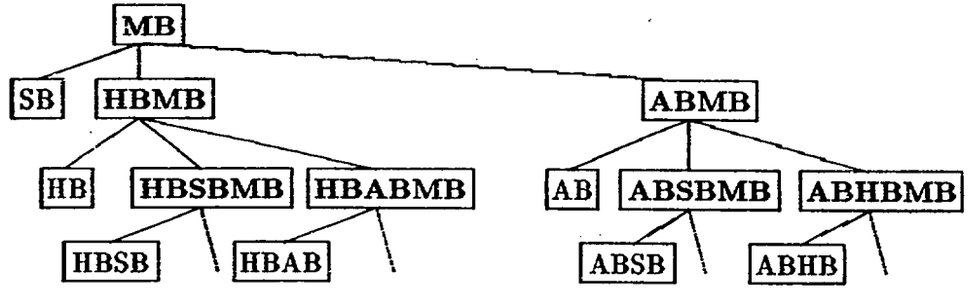

Figure 4.1: Rhet Contexts, Viewed as a Tree

Here are some examples:

```
(Assert-Axioms [MB [[P] <]]
               [Q < P R :forward R]
               [HBSB[[R] <]])
```

From this we would expect to forward chain that HBSB[Q], since HBSB inherits from MB.

```
(Assert-Axioms [MB[[P A] <]]
               [[Q ?x] < [P ?x] [R ?x]]
               [HBSB[[R] <]])
```

From this we could prove that HBSB[Q].

Another example, using assumptions:

```
(Add-EQ [MOTHER-OF OEDIPUS] [JOCASTA] :CONTEXT (OPERATOR 'SB))
```

```
(Add-EQ [MOTHER-OF OEDIPUS] [FRANCIS] :CONTEXT (OPERATOR 'HB))
(Add-EQ [WIFE-OF OEDIPUS] [JOCASTA] :CONTEXT (OPERATOR 'MB))
(Assert-Axioms
        [MB [[Not-EQ Jocasta Francis] <]]
        [[Happy ?x] < [BOUND [Wife-of ?x]]
                      [NOT [EQ? [Wife-of ?x]
                                [Mother-of ?x]]]]
        [[P ?x ?y] < [Assume [Add-EQ [Mother-of ?x] ?y]
                             [Happy ?x]]]
```

From this we can prove in HB that Oedipus is Happy, and `[P Oedipus Jocasta]` gives a contradiction. while from SB we can't tell if Oedipus is happy, and `[P Oedipus Jocasta]` returns :Unknown[1].

Additionally a program can create arbitrary context structures, and change the default context at will. The Prove and Assert-Axioms commands (among others) both take an optional context keyword for specifying what context a proof should be done in or a set of axioms should be asserted to, respectively. More details can be found in section 8.4. Note that the modal belief operators are taken relative to the default context, as are assumptions.

The current system has several limitations and makes several assumptions. First of all, the mutual belief operator expresses belief among all the agents. It is not possible to represent beliefs among a select group of agents (using the predefined belief operators). Secondly, if there is a "MB" in the operator, it had better be at the end, otherwise the system will ignore it. Finally, the system makes the reduction that if within the

[1] Actually this illustrates a shortcoming of the equality/context system as it exists: equalities cannot be retracted. So to have asserted that Oedipus believes his mother to be Francis, it cannot be revised later. This may mean that what we REALLY want to do with the belief modal operators is create a subcontext that can be destroyed when we want to retract a belief, and we recreate it with everything minus the belief we retracted. Unfortunately, given that HBSB inherits from HBMB we may have problems with such operations involving the MB modal.

- add fact [[B] <] to the new default context

I will be able to derive SB[B] and SB[A] from FOO, only SB[A] from T, and proving MB[?x] would return MB[A].

Note that an error will be signalled if an attempt is made to create any kind of mutual belief context in any UContext other than the root UContext.

Chapter 5

Other noteworthy concepts

5.1 Sets

Part of the language, but not manipulable (yet).

A set may be ordered or unordered. Basically, whenever a group of objects is to be considered together, it is usually best to treat them as sets. It is possible to use the Lisp List structure as well, using Functional Values. See 5.2 for more information on this.

[[More will come here where sets are better defined and there is something to write about]]

5.2 Functional Values

Rhet provides the builtins and lisp functions Function-Value and Set-Function-Value to establish a value for a Function Term. Normally, Function Terms may only be manipulated by equality expressions, but this facility allows them to essentialiy evaluate to something as well.

One usage of these value functions might be to provide an ordered list or other Lisp form as part of a structured type. Consider a frame for a Person, with a slot (role) for Brothers. An instance of a T-Person might be [Jack], with brothers [Bob] and [Jim]. Without fuctional values (or sets), one can make this slot equal to one of the brothers, but one cannot collect the brothers into a group (*e.g.* a list) and make the slot equal to that. Instead, we can add that:

(5.1) `(Set-Function-Value [R-Brothers Jack] '([Bob] [Jim]))`

Anything that is equal to [R-Brothers Jack] is similarly updated. Note also that the value of a Function Term will depend on context. Specifically, setting the value of a function term will not update any parent context, but will update all child contexts the term is visible in.

This specific example could better have been served by making the value a set, of course, since then we could represent the number of brothers Jack has independantly of being able to enumerate them. This is possible because the value can be any object. Since sets are terms that can be used in equality expressions, however, we would not have even needed to use this mechanism. Instead, let's consider a more general case, where we are using Rhet as a KB system intimately tied with a Lisp system. It would be possible to make the value of any slot of any frame to be a lisp function, which could be evaluated to return the information desired. Within Rhet, this might look like:

(5.2) `[Member ?y*T-Watch [Function-Value [R-Current-Wardrobe [BOB]]]]`

(5.3) `[Setvalue ?x [Function-Value [R-Current-Time ?y]]]`

Again, a Set may be more appropriate for dealing with Bob's Wardrobe, but the Current-Time on the watch will evaluate to a lisp function that can compute it.

5.3 [Cut]

The "cut" symbol. It has no effect until Rhet tries to backtrack past it, and then the prover immediately fails on the subproblem it was working on. An alternate definition: [Cut] always succeeds, and when executed, removes all choice points in the proof from the point at which the predicate which appears in the head of the axiom containing the cut was selected to the current point of the proof. To clarify:

- A goal that successfully unified with a clause that contains a cut cannot unify with any clauses that occur below the successful one.
- [Cut] prunes out all alternative solutions to goals to its left in a clause.
- [Cut] does nothing to goals to its right in a clause.

Programmers are encouraged to use *green q.v.* cuts[1] not only to help prune the search tree, thus improving computation time, but to save space since backtracking information can be curtailed. It can also be information to the compiler and user that the computation is deterministic. For example, given the following fragment:

```
[[P ?x] < [var ?x] [Cut] ...]
[[P ?x] < [atom ?x] [Cut] ...]
```

etc. where we are using some metainformation about the proof to decide which rule to invoke. It is clear that once the appropriate rule has been selected, no other is appropriate. This, then is a good place for [Cut], since it would keep us from bothering to check all the other available ways to prove [P ?x] once we have picked the right rule. Note that the addition of [Cut] in this case does not change the meaning of the program, it just makes it more efficient[2]. Another example, from Sterling and Sterling and Shapiro[Sterling and Shapiro, 1986] might be the following fragment from a sorting program:

[1] Logic programming purists actually don't believe having cut is a good idea. This is because [Cut] has a procedural semantics only.

[2] In the PROLOG literature, such a cut is known as a *green cut*. A *red cut* would be one that, if omitted, would change the solutions found (rather than, say, the order of the solutions).

```
[[sort ?x ?y] < [append ?a (?x1 ?y1 .  ?b) ?x]
                [greater ?x1 ?y1]
                [Cut]
                [append ?a (?y1 ?x1 .  ?b) ?x2]
                [sort ?x2 ?y]]
```

The interesting thing to note about this program is that once we have determined that two elements need to be swapped, we can execute the [Cut]. There probably is more than one way to sort a particular list, but there is only one sorted list to be output. This keeps us from looking at all the possible ways of sorting a list, and lets us concentrate on the actual sort for a particular approach.

Another use of [Cut] might be to force tail recursion optimization to apply in certain cases. That is, given:

(5.4) `[[P (?x*T-Atom . ?y*T-List)] < [Q ?x] [P ?y]]`

(5.5) `[[P ()] < [Q :A]]`

Rhet can optimize the tail recursion, since ?y cannot be empty. But, given:

(5.6) `[[P ?x] < [frotz ?x ?y] [P ?y]]`

we cannot tell that there is not more than one "choice point" for binding ?y and so cannot optimize. Inserting a [CUT] before this clause will allow optimization, although it is possible that this will change the semantics of the program.

Chapter 6

Performance Issues

Naturally, we have done all we can to make Rhet as efficient as possible, given other constraints, *e.g.* on development time, the point in the development cycle we are now in, *etc.* (Rhet will be initially released without a true axiom compiler, which will be one of the first things added). In many cases we have sacrificed certain 'features' of PROLOG to gain this efficiency. For example, note that Axiom order in PROLOG is extremely important. Rhet normally feels free to optimize axiom ordering as needed, esp. in compiled code. For instance, FC axioms can have their RHS reordered at will, since KB lookup order does not involve a semantic change. Usually all tests involving (only) local variables will be moved to the front of the RHS, and the trigger will be eliminated from the RHS if it appears (since the axiom would not have fired unless the trigger had been true). BC axioms do change semantics, or at least, the order of proofs generated. At the moment, they are not optimized, although in the future variable declarations as in Warren [Warren, 1977a] may be added, axioms may be defined to be evaluated in parallel, *etc.*. For this reason, we have left evaluation order of axioms and the evaluation order of clauses within axioms undefined. The And and Or builtins[1], however,

[1] See section 7.6

are defined to short-circuit, and therefore should be used when clausal ordering is important.

6.1 User-Level Optimizations

Nonetheless, there are many things the user can do to improve the execution speed of their program. I summarize a few here. More information of this sort can be gleaned from chapter 13 of [Sterling and Shapiro, 1986].

- The earlier you fail in a particular rule or set of rules, the better pruned the search tree becomes. Try to fail as early as possible.
- Take advantage of tail recursion where possible. Iteration costs less in space than true recursion does[2].
- Use green cuts to improve the efficiency of clause selection, but not to the extent that readability suffers.
- Explicit conditions can make your program more deterministic. (*E.g.* `[VAR ?X]` is an explicit condition). Rhet will eventually (when the compiler is installed) use these to optimize clausal selection.
- Use the type system to it's full capabilities. While setting up a rich type hierarchy can be expensive in compute time, the world with these types built-in can be saved, and proofs using typed variables and objects are much quicker. Again, you are pruning the search space.
- Avoid overuse of the context and equality system. The deeper in the context tree you are, the more expensive adding equalities become. On the other hand, looking up equalities is always efficient. Inequality lookup, on the other hand, may not be.

[2]Currently this optimization may be unimplemented, however, it is still good practice to get into.

- Avoid declaring certain Lisp functions as builtins. Adding type information, calculating the inequivalence class of an object, *etc.* are all very expensive to compute, and should be avoided within your axioms. Some may interact with the prover in such a way as to invalidate the proof being attempted.
- Utilize the Post-Constraint Mechanism wherever it makes sense. It is often considerably less work for the prover to constrain a variable and later do the proof for a particular binding than to have to backtrack. The Post builtin is typically indicated when global[3] variables are used in the RHS of an axiom.

6.2 Program-Level Optimizations

I describe here, for completeness, some of the things Rhet does to optimize FC and BC proofs that have not already been described, and summarize other optimizations.

- The order axioms or facts have been asserted to Rhet is ignored. For practical purposes, all facts that will unify with a goal are retrieved in parallel, all proofs of a goal are attempted in parallel. For this reason, side effects, and ordering requirements need to be made explicit, normally using the And and Or operators, which are defined to evaluate sequentially, and short-circuit[4].
- Reread the above item. It's important.
- Rhet does not attempt to reprove the trigger of a FC axiom. However, by leaving it out of the RHS, it will not be included in the justification list. Normally, this will not be what is wanted, so the trigger should appear both as a trigger and on the RHS of a FC axiom.
- Rhet, after triggering a particular FC axiom, will first check (if global variables are not involved) that the consequent is not already true.

[3] A global variable is one that is not bound by axiom invocation. It may either be a variable that is present in the RHS of an axiom and not the LHS, or a variable in the goal, and presented to Prove.

[4] Side effects are not currently supported intelligently. Avoid them.

- Rhet will do intelligent backtracking [Bruynooghe and Pereira, 1984][Cox, 1984] to a limited extent in it's interpreter, and will do so to a much greater extent when the axiom compiler is implemented. Currently, Rhet is able to avoid reinterpreting deterministic clauses, and to return a proof back to the last state where a global variable involved in the currently failing subproof was bound, though it will always choose the closest such binding[5]. The games played with the stack are a simulation of Scheme continuations into inactive closures, and was influenced by [Kahn and Carlsson, 1984]. This cleverness is more general than the Post-Constraint Mechanism, but is not as efficient, in most cases. In fact, the two work together, since Post will prevent a bad binding for a variable in the first place, and if no binding can be found, the backtracking mechanism will reset Rhet to the point the Posting was performed, which may be a considerable savings over the closest non-deterministic node in the current proof tree.

- Rhet will save and reuse stack when deterministic proofs are done, as in [Mellish, 1982], [Bruynooghe, 1982].

- Goal caching is currently under development. See [Fagin, 1984], for more information on this technique.

[5] eventually, we should be able to choose the particular variable that is the culprit, rather that using the closest bound one

Part II

Rhet Reference Manual

Chapter 7

The Language

7.1 Conventions

This document (and Rhet) follows the following basic conventions. Rhet expressions appear inside of square brackets, *i.e.* "[" and "]" to distinguish them from Lisp lists, which use the normal parenthesis notation. Inside of a Rhet expression, anything which looks like a normal Lisp atom is actually a Rhet term. Real Lisp atoms, should they be needed, are preceded by a colon (:), unless they appear inside of a Lisp list, in which case normal Lisp syntax prevails. Rhet variables always have a question mark (?) as their first character. Thus

(7.1) `[P ?x (A B [C]) :D]`

is a Rhet expression whose head (or Predicate Name following the usage in table 7.1) is the Rhet term P; and whose arguments are, in order, a Rhet variable, a Lisp list (consisting of two Lisp atoms and a Rhet Predicate) and a Lisp atom.

7.2 Syntax

The six major classes of expressions in this language are Terms, Atomic Formulas, Forms, Facts, Basic Expressions and Axioms. The syntax for these classes are given by the BNF rules in table 7.1. Note that in this table quotes ("") delimit an actual string or token expected by the reader (thus the rule for a Lisp-Atom indicates that an actual typed-in literal colon (:) must be followed by a constant); "|" is an or sign, "*" represents Kleene Closure, and "+" is like Kleene Closure, but requires at least one occurrence.

A Fact is something that Rhet will allow to be asserted as true. Basically any Rhet expression that does not contain variables or a RHS if an index is present is potentially a Fact. Thus,

(7.2) `[P A]`

is potentially a Fact (it could be asserted), as is

(7.3) `[P :F (Foo Bar)]`

and

(7.4) `[[P B] <]`

is certainly a Fact (the presence of an index removes ambiguity) but neither

(7.5) `[P ?x]`

nor

(7.6) `[[P C] < [P D]]`

are since the former includes a variable, and the latter has a RHS (symbols beginning with '<' are considered an index). Forms include Facts, but also include Rhet expressions with variables. Forms cannot have an index. Thus

(7.7) `[P ?x]`

is a Form, as is

(7.8) `[Q [P ?x]]`,

but

(7.9) `[[P ?z] <]`

isn't since it includes an index. Expressions that include an index and are not a Fact (that is, they either have variables, or a RHS) are Axioms.

(7.10) `[[P ?x] <]`

is an axiom, as is

(7.11) `[[P C] < [P D]]`.

Lisp expressions may be involved in Forms, Axioms or Facts, but never in Basic Expressions.

(7.12) `[P :A]`

<Set>	::=	“” (<Axiom> | <Term>)* “”
<Fact>	::=	<Predicate Name> | “[” <Predicate Name> (<Lisp Expression> | <Basic Expression>)* “]”
<Indexed-Fact>	::=	“[” <Predicate Name> (<Lisp Expression> | <Basic Expression>)* <Index> “]” | “[” <Fact> <Index> “]”
<Function Term>	::=	“[” <Function Name> <Term>* “]”
<Axiom>	::=	“[” <Conclusion> <Index> <Premise>* <Keyword>“]”
<Conclusion>	::=	<Atomic Formula>
<Premise>	::=	<Atomic Formula> | “CUT”
<Atomic Formula>	::=	“[” <Predicate Name> <<Term> | <Form>>* “]” | <Function Term> “[” NOT <Predicate Name> < <Term> | <Form> >* “]” | “[” NOT <Atomic Formula> “]”
<Form>	::=	<Fact> | <Function Term> | “[” < <Term> | <Form> >+ “]”

Table 7.1: Syntax Table — Part I

<Basic Expression>	::=	<Constant> | "[" <Predicate Name> <Basic Expression>* "]" | "[" NOT <Basic Expression> "]"
<Index>	::=	"<" <String-Constant>
<Predicate Name>	::=	<Constant>
<Function Name>	::=	<Constant>
<Term>	::=	<Constant> | <Variable> | <Fact> | <List> | <Atomic Formula> | <Lisp-Atom> | <Function Term>
<Lisp Expression>	::=	<Lisp-Atom> | <Restricted-List>
<List>	::=	"(" <Term>* ")" | "(" <Term>+ "." <Term> ")" | "NIL"
<Restricted List>	::=	"(" <Constant>* ")" | "(" <Constant>+ "." (<Constant> | <Restricted List>) ")" | "NIL"
<Keyword>	::=	< ":Forward" < <Atomic Formula>* | ":All" > > | ∈∈
<Type-Expr>	::=	<Type> | "(" <Type>* ")" | "(" <Type>* - <Type>* ")"

Table 7.2: Syntax Table — Part II

<Constant>	::=	<Literal Atom>
<Lisp-Atom>	::=	":" <Constant>
<Variable>	::=	"?" <Literal Atom> | "?" <Literal Atom> * < Type-Expr > | "[" "ANY" <Variable> <Premise>+"]"

Table 7.3: Syntax Table — Part III

is a Form, or a Fact, but

(7.13) `[P [C (D E)]]`

may only be a Form, since the subexpression is not a Basic Expression. Note that Function Terms are syntactically indistinguishable from certain Facts, the difference being that instead of a Predicate Name in the first position of the expression, the name of a Builtin function or declared Lisp function will be used.

Note that what is considered to be an atom is the same as a keyword in Common Lisp. Further, all atomic (non-string) terms are converted internally to upper-case, just as in Common Lisp. For example,

(7.14) `[P-x a-b-c ?x*foo]`

and

(7.15) `[p-x A-B-C ?x*FOO]`

are internally represented identically. Normally, output is in upper case only, with the exception of variables which do not force case (that is, ?x is distinct from ?X). This document will follow the convention of using upper-case names for atoms, and lower-case for variables (following the '?' symbol). Indexes, on the other hand are considered to be strings, and therefore case is preserved. Thus, "<foo" is not the same index as "<Foo". All indexes are required to begin with the character '<' to distinguish them from forms when parsing an axiom vs. a form.

An example of an axiom is:

(7.16) `[[P ?x] <1 [Q ?x]]`

where "[P ?x]" is the <conclusion>, "<1" is the index, and "[Q ?x]" is a simple <list of premises>. This statement is interpreted as follows: the assertion named[1] "<1" signifies that for any x, [Q x] implies [P x]. Or, alternately, to prove [P x] for any x, try to prove [Q x].

An example of a set[2] is:

(7.17) `{ [[A] (B C) [[Foo ?x] < [Bar ?x]]] }`

which would be a set consisting of a Form, a List, and an Axiom.

7.2.1 Special Symbols

The Rhet system uses several special symbols which should not be used for other purposes. They are defined to be reader macro characters, and loading Rhet may make destructive changes to the default readtable[3].

[1] Actually, 'named' may be a misnomer, since the index can be non-unique.

[2] Sets, while they are defined and recognized, currently cannot be manipulated.

[3] This is an installation option, see appendix of [Miller, 1989].

? in the printed representation indicates a variable. It will cause the atom following it to be expanded into the internal variable format on input.

***** in the printed representation indicates that the expression following it is a type. This is true only in axioms and constants. The symbol can be used freely in Lisp code.

xB operator in the printed representation indicates a belief operator. This is only true in axioms and constants. These symbols can be used freely in Lisp code without special interpretation. Examples of belief operators are "MB" and "SBHBMB".

; in source code is considered a comment start, and text between it and the end of the line is not interpreted.

#| in source code is the beginning of a structured comment. Everything between this and the end structured comment character is ignored. This is usually written as #|| since the Lisp reader treats |anything at all| as a single token, so in this manner a comment is treated as one token to be ignored.

|# in source code is the end of a structured comment. This is usually written as ||# as above.

[in source code is considered the beginning of some Rhet structure.

] in source code is considered the end of some Rhet structure.

#[in source code is considered the beginning of a Rhet binding environment. This is needed when more than one Rhet structure or variable will appear at top level, but the variables in these structures are meant to be the same if they print the same. For example, when doing a Prove-All where each clause may refer to the same variable ?x, as in:

(7.18) `(Prove-All #[[IS-STEP-IN-SOME-PLAN [C-GO ?X ?Y] ?Z*T-PLANS] [PRINT-PLAN ?Z]#])`

the #[#] forms make sure the references to ?z are the same in the two goals. Otherwise they would be assumed to be different, since they are otherwise indepandant clauses. In this case, we could also have used the **And** builtin to get around this problem, however examine the following:

(7.19)
```
#[ ; these force the read of the defrhetpred into a single variable binding environment.
(DEFRHETPRED IS-PLAN-STEP-LF (&BOUND ?STEP*T-ACTION ?PLAN*T-PLANS)
     "Like IS-PLAN-STEP, but a lispfunction, so in theory faster.
      Only works if ?step is not a form with variables!"
     (DECLARE (MONOTONIC))

     (MEMBER ?STEP (FUNCTION-VALUE [F-PLAN-STEPS ?PLAN])))
#] ; close the special reading environment.
```

Since this DefRhetPred is for a lispfn, we have to use these reader symbols to make sure the reference to the variable in the arglist is the same as the one in the embedded Rhet form. Had no Rhet forms been embedded, it would not have been needed: DefRhetPred is smart enough to handle unimbedded references to arglist variables itself.

] End of a Rhet binding environment.

{ is a set constructor, and begins a sequence to be taken as denoting a set.

} ends a set constuctor.

7.3 Typed Terms

The type of a variable is indicated by appending a suffix to the variable indicating its type. Thus ?x*CAT names a variable ?x that is of type *CAT. The variable ?x*CAT will unify only with terms that are compatible with the type *CAT. The type of a variable is by default *T-U, the most general type for Rhet objects. This cannot be bound to a Lisp object, however. Inside of a List, the default type of a Rhet variable is *T-Lisp which can only be bound to Lisp objects.

Types should be viewed as sets, and no restrictions are assumed as to whether sets are disjoint, mutually exclusive, or wholly contained by each other. This information is specified by the user by declaring the type hierarchy as defined in section 8.11. Instances of types may be specified with the following Lisp functions (which are also available as Builtins):

Contrast to HORNE

Itype <*Typename*> &Rest <*Individual*>
which asserts that the individual(s) are of the indicated immediate type, *e.g.*, (Itype 'Cat [A]) asserts that the constant [A] is of type CAT, and is not in any known subset.

Was TYPE

Utype <*Typename*> &Rest <*Individual*>
This asserts that the individual is of the indicated type, *e.g.*, (UTYPE 'CAT [A]) asserts that the constant [A] is of type CAT, but may be a subtype, *e.g.* KITTEN.

New

Dtype <*Individual*> &Rest <*Typename*>
This is similar to the Utype assertion, above, but also indicates that the individual cannot possibly be equal to any other Dtyped individual *of the type typename*. See also 3.2.

7.4 Structures in Rhet

The Rhet system supports reasoning about structured types (see section 3.3.1 of the introduction). The following naming conventions, are used to distinguish the different kinds of objects and are observed by the Rhet system.

T- ...— a type name

R- ...— a rolename

F- ...— the function named by a rolename

C- ...— a constructor function

To define a subtype with roles, there are two options, depending on whether the objects of the new type are fully determined by the set of roles defined. Both of these enforce the restriction that the new type must be a subtype of an existing type. A type *T-U is predefined as the root of the Rhet type hierarchy[4]. See section 8.12.2 for a description of how to retrieve role information about objects. See section 8.12.1 for a description of how to define roles on the type hierarchy.

7.5 Reasoning Modes

The unifier in Rhet has been augmented to allow two types of special unification dealing with equality and restricted variables.

7.5.1 Equality

The unification algorithm of Rhet has been modified so that when terms do not unify they can be matched by proving that the terms are equal. Any variables in the terms matched will be bound as needed to establish the equality. Equality statements are added to the system either by calling the Lisp function or by using the builtin Add-EQ[5]. For example:

(7.20) (Add-EQ [president USA] [Ronald-Reagan])

expresses a fact that is well known to most Americans. The axiom

(7.21) (Add-EQ [add-zero 1] [1])

[4]The actual root is type *T-Anything which includes, besides *T-U, type *T-Lisp, a special type for Lisp objects, and its children.

[5]Add-EQ may not appear in the LHS of an axiom, thus its presence as a separate Lisp function rather than as something Assert-Axioms might add.

expresses an infinite class of equalities. For example, `[add-zero [add-zero 1]]` equals `[1]`, as does `[add-zero [add-zero [add-zero 1]]]`, and so on.

The system provides, in an efficient manner, complete reasoning about fully grounded terms (*i.e.*, terms that contain no variables). The system will allow variables in queries (which may be bound to establish equalities).

The information derived from the Add-EQ axioms that are asserted is stored on a pre-computed table which is updated as Add-EQ axioms are added. Add-EQ axioms may not be deleted, however, since Add-EQ axioms may be added relative to a context, it is possible to pop the entire context.

7.5.2 The Post-Constraint Mechanism

Rhet allows the user to specify that the proof of an atomic formula be delayed until the terms in it are completely bound. The user does this by enclosing the atomic formula within the function `[POST]`, as in the axiom:

(7.22) `[[F ?x] < [POST [MEMBER ?x (a very long list)]] [G ?x]]`

POST takes an atomic formula as an argument. If the formula is grounded then the proof proceeds as usual. Otherwise the variables in the formula are bound to a function which restricts its value and the proof proceeds as though the proof of the formula succeeded.

Restrictions on variables are implemented by setting a constraint on the variable that is presented as:

(7.23) `[Any ?newvar [constraint ?newvar]].`

Thus, give the above axiom, if we queried `[F ?s]`, the `[POST]` mechanism would constrain `?s` to

(7.24) `[Any ?s0001 [MEMBER ?s0001 (a very long list)]].`

This use of the special form Any is similar to the omega form used in Kornfeld [Kornfeld, 1983].

The Rhet unifier has been modified so that it knows about constrained variables. A term printed with the form `[Any ?x [R ?x]]` will unify with any term that satisfies the constraint `[R ?x]`. Again using the above axiom: after the POST succeeds, the proof continues with the subgoal

(7.25) `[G [Any ?s0001 [MEMBER ?s0001 (a very ...)]]].`

Now suppose that `[G e]` is true. Then we can unify these two literals if we can prove

(7.26) `[MEMBER e (a very long list)].`

Note that the constraint will be queried only once its variable is bound. Thus if `[G ?c]` were true above, the unification would succeed and

(7.27) `[F [Any ?s0001 [MEMBER ?s0001 (a very long list)]]]`

would be returned as the result of the proof. If `[G [fn ?c]]` were true instead, a recursive proof testing whether `[MEMBER [fn ?c] (a very long list)]` would be done and, if successful, the final result of the proof would be

(7.28) `[F [fn [Any ?z [MEMBER [fn ?z] (a very long list)]]]].`

NB: Certain builtins automatically post constraints on variables as needed. For example, if Rhet tries to prove `[Distinct ?x B]`, it does not bind ?x to something that happens to not be `[EQ?]` to `[B]`, but rather constrains ?x by changing it to: `[Any ?x [Distinct ?x B]]`, which allows us to check for distinctness whenever ?x is actually bound. This avoids our having to pick a particular binding for ?x at the current time, which would likely force us to backtrack later should our guess be wrong.

7.5.3 Backward Chaining

This has adequately been described in section 2.1.

7.5.3.1 Defining Backward Production Axioms

Backward Chaining Axioms are added to the system via the Assert-Axioms Lisp function. As an example,

(7.29) (Assert-Axioms [[e ?f] <j])

would allow us to prove [e foo], while

(7.30) (Assert-Axioms [[w ?d] <j [r ?d]])

(7.31) (Assert-Axioms [[r d] <j])

would allow us to prove [w d]. For convenience, Assert-Axioms may take several Rhet axioms. Thus the above two assertions could have been made by

(7.32) (Assert-Axioms [[w ?d] <j [r ?d]] [[r d] <j]).

7.5.4 Forward Chaining

The prover has a forward production system in which the addition of new axioms adds new facts that are implied by the existing axioms. The general form of a forward axiom is as follows:

[[conclusion] <index <[condition]>* :forward <[trigger]>*]

Note that this is just a regular horn clause with one of the conditions being the keyword :forward followed by the triggers or the keyword :All, or nothing (which is equivalent to specifying :All).

After a Rhet axiom is added to the database it is checked to see if it matches any trigger pattern. A trigger must be an atomic formula, but cannot be a Lisp predicate. If it matches, then using the binding list of the match the system tries to show that the conditions associated with the trigger are in the database. Note that the system does not try to prove the conditions (unless specified), but simply checks that they are in the database. If all the conditions can be shown to be in the database then the conclusion is added to the Rhet axiom list using the bindings collected in the process. Lisp predicates can be used in the conditions and in the conclusion, where they are called as in the backwards chaining system. The value returned by a Lisp predicate in the conclusion is ignored[6]. In adding a conclusion another trigger may be fired. To prevent infinite looping the forward chaining system will not add axioms that are already in the database. Rhet will do a simple consistency check before adding a forward chained assertion, namely, that it's inverse is not already asserted. This will not necessarily prevent contradictions from being discovered later, since the deductive closure is not kept.

7.5.4.1 Defining Forward Production Axioms

Forward Chaining axioms are added just like normal backward chaining axioms are, *i.e.* via Assert-Axioms. For example,

(7.33) `(Assert-Axioms [[w ?d] <r [r ?d] :forward [r ?d]])`

(7.34) `(Assert-Axioms [[r d] <s])`

will result in the derived fact

[6]This would only be of use if the predicate had side-effects

(7.35) `[[w d] <r]`

being added to the database, with the above statements for support. Using the atom :All for the trigger adds a separate forward-chaining axiom for each of the atomic formulas in the condition list with that condition as the trigger[7]. Thus each of the conditions is a trigger, *e.g.*,

(7.36) `(Assert-Axioms [[eq ?y ?z] <1 [eq ?y ?x] [eq ?x ?z] :forward :all])`

adds the following forward chaining axioms to the system:

(7.37) `[[eq ?y ?z] <1 [eq ?y ?x] [eq ?x ?z] :forward [eq ?y ?x]]`

(7.38) `[[eq ?y ?z] <1 [eq ?y ?x] [eq ?x ?z] :forward [eq ?x ?z]]`

Given these, the following additions:

(7.39) `(Assert-Axioms [[eq w e] <1])`

and

(7.40) `(Assert-Axioms [[eq r w] <1])`

causes the axiom

(7.41) `[[eq r e] <1]`

[7]This is also the default: that is if no trigger is specified, :All is assumed.

to be added to the system.

Naturally, a Lispfn may occupy the position of the predicate name in any of the conditions. The Lispfn succeeds if it returns a non Nil value.

The position of the predicate name may also be occupied by the atom :Prove whose argument is an atomic formula. This allows any of the conditions to call the Reasoner to prove the condition. (Note that normally conditions are not proved but just shown to be in the data base). The condition is true if the atomic formula can proven by the Reasoner. Any variables bound in the proof will be passed on to the next condition.

The system does perform truth maintenance; *i.e.*, axioms entered into the data base due to a forward-chaining axiom are removed when the axiom is removed, if it has no other support.

7.6 Built-In Predicates

This section documents the built-in predicates (referred to as Builtins) that are already defined in Rhet. Many also are defined as Lisp functions that can be called directly. For instance Assert-Axioms, the primary way of adding axioms to the system, is both a Lisp function (as defined above) and a Builtin. Some of the following builtins are described as being assertable. This means that it is legal for them to appear inside an Assert-Fact, Assert-Axioms, or on the LHS of a Forward Chaining axiom.

[Add-EQ Term1 Term2**]** EQ in HORNE

Both terms must be fully grounded. This predicate will add the equality specified. Note that the context the equality is added in may be explicitly specified by Assert-Axioms, or the modal operators if this is in a horn clause. Since equalities cannot be retracted, the Assume builtin is typically used in conjuction with it. Note that this function is also available directly from Lisp.

[Add-InEQ Term1 Term2**]** New

Both terms must be fully grounded. This predicate will add the inequality specified. Note that the

context the inequality is added in may be explicitly specified by Assert-Axioms, or the modal operators if this is on the RHS of a horn clause. Like equalities added via the Add-EQ function or builtin, inequalties cannot be retracted. Note that this function is also available directly from Lisp.

Was Role

[Add-Role <Instance> *&Rest* <<Role-Name> <Value>>*]

Adds that object <Instance> has role <Role-Name> with value <value>. All arguments must be fully grounded. Note that Role-Name is expected to be a Lisp atom, *e.g.* :R-Name.

New

[And <Form>*]

Succeeds only if all forms can be proven. `[AND]` succeeds. If `[CUT]` is encountered as a Form, backtracking past it cuts out of the entire AND, but not necessarily out of the entire rule. A singular AND on the RHS of a horn clause has the same semantics as a normal horn clause's RHS. Note that And will evaluate it's arguments specifically in the order supplied, and will short-circuit evaluation as needed. Thus the clause `[And [P ?x] [Q ?x]]` will not cause evaluation of `[Q ?x]` if `[P ?x]` cannot be proven. This builtin is assertable; all of the Forms in the argument list must themselves be assertable.

New
Current Status: Untested

[And* <Form>*]

A macro form for `[AND [CUT] Form1 [CUT] Form2 ...[CUT] FormN]`. That is, any backtracking needed at all inside of the AND* causes it to fail. This builtin is assertable; it acts just like And does in an assertion.

Enhanced
Current Status:
Dangerous

[Assert-Axioms <Axiom1> ...<AxiomN> *&key* Context Justification]

Adds the specified axioms to the data base for the specified predicate. It returns non-nil, thus succeeds when found on the RHS of a horn clause. All logic variables in the new axioms that appear outside the axiom assertion in the current environment will be replaced by their values before the new axioms are added. If the context parameter is specified, the string passed will indicate to the system which context facts should be asserted relative to. It will otherwise default to the default context, which the user can set programmatically. Note that this is one of the rare Rhet predicates that is available directly from Lisp, as Assert-Axioms. Note that in general the facts passed to Assert-Fact may not be builtins, however,

certain builtins and lispfns that have been declared to be assertable (see 8.7.3 for more information on assertable lispfns) may appear. For example, And may appear here, though it isn't really necessary; Format may appear, again, having the same functionality as if it appeared without being wrapped by the Assert-Fact, and modal operators may appear. Assertable builtins are documented as such.

[Assert-Fact <Fact1> ...<FactN> *&key* Context Justification**]** New

Adds the specified axioms to the data base for the specified predicate. It returns non-nil, thus succeeds when found on the RHS of a horn clause. All logic variables in the new facts that are bound in the current environment will be replaced by their values before the new facts are added. If the context parameter is specified, the string passed will indicate to the system which context facts should be asserted relative to. It will otherwise default to the default context, which the user can set programmatically. A runtime error is generated should any variables remain unbound. Note that in general the facts passed to Assert-Fact may not be builtins, however, certain builtins and lispfns that have been declared to be assertable (see 8.7.3 for more information on assertable lispfns) may appear. For example, And may appear here, though it isn't really necessary; Format may appear, again, having the same functionality as if it appeared without being wrapped by the Assert-Fact, and modal operators may appear. Assertable builtins are documented as such.

Current Status:
May only be called with any embedded vars in the Facts fully bound.

[Assert-Relations <Instance> Relation-Keyword *&Optional* (Accessor-String (String Relation-Keyword))**]** New

This builtin presumes a specific usage for relations: that of constructor functions for other types. For each of the relations of Relation-Keyword, it is generated, and set (via Add-Eq) to an accessor, which is by default `[relation-# ?self]`, that is, given 3 STEPS being defined on type Eat-Lunch-Plan, (locally or inherited), and having constructed an instance of Eat-Lunch-Plan, say EL1, then `[Assert-Relations EL1 :steps]` will succeed and generate three accessors, `[Steps-1 EL1]`, `[Steps-2 EL1]`, and `[Steps-3 EL1]`, each `[EQ?]` to whatever was on the STEPS relation in those positions, with ?Self appropriately bound. If this form is a constructor function, the construction is done at this time as well (at least until Rhet can otherwise automatically handle it). A more complete example of possible use is given in section 8.38. A special usage is when Accessor-String is passed as :T. In this case, the relation defined

should itself be an ALIST, whose key is an atom whose printname will be used as the accessor-string, and whose value will be the interesting constructor function.

For example, the former usage would have a **Define-Subtype** form that looked like:

(7.42)
```
(Define-Subtype 'T-HACK-1 'T-U
                :Roles '((R-ACTOR T-ANIMATE))
                :Relations '((:STEPS [C-EAT-PIZZA [F-ACTOR ?self]]
                                     [C-DRINK-BEER [F-ACTOR ?self]]
                                     [C-HACK-COMPUTER [F-ACTOR ?self]])))
```

while the latter would be like:

(7.43)
```
(Define-Subtype 'T-HACK-2 'T-U
                :Roles '((R-ACTOR T-ANIMATE))
                :Relations '((:STEPS (STEP-1 . [C-EAT-PIZZA [F-ACTOR ?self]])
                                     (STEP-3 . [C-DRINK-BEER [F-ACTOR ?self]])
                                     (STEP-5 . [C-HACK-COMPUTER [F-ACTOR ?self]]))))
```

The advantage of the latter usage is mainly so the steps can be sparsely named; subtypes could then supply other steps in the sequence, or specialize steps created by the parent type. *I.e.:*

(7.44)
```
(Define-Subtype 'T-HACK-LISPM 'T-HACK-2
                :Relations '((:STEPS (STEP-1 . [C-EAT-GOURMET-PIZZA [F-ACTOR ?self]])
                                     (STEP-2 . [C-CHEW-HUNAN-PEPPER [F-ACTOR ?self]])
                                     (STEP-3 . [C-DRINK-MOLSON-BEER [F-ACTOR ?self]])
                                     (STEP-4 . [C-READ-NETNEWS [F-ACTOR ?self]])
                                     (STEP-5 . [C-HACK-LISPM [F-ACTOR ?self]]))))
```

Remember that [Assert-Relations] supports a particular kind of usage of the relations field; other usages are possible.

[Assume <Fact> <Form>*] — New

Creates a subcontext to the default one and asserts the Fact in this subcontext. It then attempt to prove each of the Forms, which can be an arbitrary series of clauses, as on the RHS of a horn clause. The ASSUME succeeds or fails according to the same rules as an axiom's RHS allowing the LHS to be considered proved. Note that ASSUME, like all predicates, can be wrapped in one or more modal operators, with the exception of an MB operator[8]. Note also that while ASSUME creates a context, it is impossible to move to this context, or use it as an argument to any function that takes a context as an argument[9].

[Atom? <Term>]

Succeeds if Term is an atom.

[Bagof Var1 Form Var2] — New; Current Status: Unimplemented

This is like [SETALL], but it allows duplicates in Var1. This will typically be faster than SETALL. It succeeds if Var1 is set to the list of all non-unique assignments to var2 that satisfy Form[10]. For example, [Bagof ?x [P ?y ?z] ?z] would set ?x to a list of all things that have been asserted as arguments to predicate P that make it true. For example, given [P A B], [P B C], and [P D C] ?x would be bound to ([B] [C] [C]).

[Bound ?x] — Differs

Succeeds only if ?x is a bound variable. It fails on any other term. For example, [bound ?x], where ?x

[8] *I.e.* MB[ASSUME [foo] [bar]] which would read "assuming that it is mutually believed that foo, can bar be proved?" While supporting such a beast would be nice, the user must handle such cases manually, somehow.

[9] clear?

[10] By this we mean that the variable Var2 is assigned to a ground term, such that the Form with Var2 so bound is found in the KB directly, as opposed to via backward chaining.

is bound to the Fact FOO succeeds, while [bound boy] or [bound ?x] fails. Distinguish this from [NOT [VAR ?x]].

New

[Call Lisp-Expression]

Calls the interpreter on the Lisp expression passed. This allows the user to call arbitrary Lisp expressions without having to use Declare-Lispfn on them. Note that the argument must be a Lisp EXPRESSION not a function[11]! (*E.g.* (FOO X) is OK, #'FOO isn't). In other words, pass a Lisp list. Unbound variables will remain as variable structures, so the user must either handle these[12], or make sure the variables are bound (using, *e.g.*, Bound) before allowing a Call to be processed.

New

[Cond Condbody]

What you would expect if you were writing in Lisp. The predicate, however, must be a Rhet predicate (or an appropriately declared Lisp predicate) and the first provable predicate form encountered causes the associated action forms to be proved. If any action fails, Cond immediately fails, but if a pred fails, Cond tries the next predicate in its list. Cond succeeds if some predicate succeeds, and no associated executed action fails. If no successful pred is found, Cond fails. Note that for backtracking purposes,

(7.45)
```
[COND ([pred1]
       [actionset1-1]
       [actionset1-2] ...)
      ([pred2]
       [actionset2-1] ...)
      ...]
```

[11]This is, in fact true, of all Rhet functions documented as taking a Lisp expression as an argument. This allows the Rhet system to bind variables in the arguments of the function appropriately, which it could not do if it were handed a compiled object. If the user is concerned with runtime efficiency, or wants to be able to backtrack, they should get a copy of the Programmer's Manual and write a Builtin instead.

[12]via Setvalue and Genvalue

acts as follows:

```
(7.46)  [OR [AND [pred1]
                 [CUT]
                 [actionset1-1]
                 [actionset1-2] ...]
            [CUT]
            [AND [pred2]
                 [CUT]
                 [actionset2-1] ...]
            [CUT]
            ...]
```

The idea is that once a pred has succeeded, the axiom can be considered to consist only of the actions associated with that pred. No other pred will be tested on backtracking, though actions can be backtracked through[13]. In PROLOG, separate clauses with a cut after their first form (a typical idiom) give this behaviour, but since RHET does not define execution order for defined clauses, Cond is used in its stead.

Note that the [Win] and [Fail] builtins can be used to good effect inside of a Cond.

[Cut]

The Cut symbol. It has no effect until Rhet tries to backtrack past it, and then the prover immediately fails on the subproblem it was working on. An alternate definition: cut always succeeds, and when executed, removes all choice points in the proof from the point at which the predicate which appears in the head of the axiom containing the cut was selected to the current point of the proof.

Current Status: Should Work in Version 16.0

[13]Thus, it would not be a good idea to use global variables in the predicate, since only one binding will be made, even in a Prove-All.

[Distinct Term1 Term2**]**
Succeeds if both terms are fully grounded, but to different atoms. If a term is not fully grounded, this posts a constraint on the variable(s) and succeeds.

New

[Dtype <Type Descriptor> *&Rest* <Form>**]**
Sets the Form(s) (any Phet object) to be of the type described, as for **Utype**. Note that this type is NOT defined to be the immediate type. It is legal to later change the type (via **Utype** or **Itype**) to a more specific type. It signals an error to set the type of an atom to be something that is not a subtype of it's existing type, or to be anything other than it's immediate type should that have been declared (see **Itype**). See also **Utype**. The difference between **Dtype** and **Utype** is that the former also declares the Form to be unequal (see **ADD-InEQ**) to any other Form that is **Dtyped** for this specific type. This builtin is assertable.

Note `[EQ?]` vs. `[Add-EQ]`.

[EQ? Term1 Term2**]**
Succeeds if Term1 equals Term2 (*i.e.*, they unify). Posts constraints as needed.

[Fail]
This predicate is always false.

Current Status:
Builtin unavailable

[Find-Facts <Atomic Formula>**]**
Same as the Lisp function find-facts in Section 8.1.2.

New
Current Status:
Known to work only if no form contains a variable not in the variable list, and all variables are used in Form1.

[Forall <Variable / list> Form1 FormN*****]
Succeeds if for all variables in Variable / list (can be an atomic variable) as constrained by Form1, (that is, all possible assigaments or constraints on the variables imposed by Form1) each form of FormN* succeeds. For example, `[Forall ?y [P ?y] [Z ?y]]` says that for all assignments to ?y (*e.g.* if `[P A]` is in the KB, then A) Z of it must be true for the form to succeed.

In general this semantic is a little flakey, because `[Forall (?y ?z) [P ?y] [Z ?z]]` is closer to

$$\forall y\ P(y) \Rightarrow \exists z\ Z(z) \tag{7.1}$$

when we probably meant

$$\forall y, z \; P(y) \Rightarrow Z(z). \tag{7.2}$$

We are currently looking at expanding Forall to have the more intuitive definition, and adding an Exists builtin so such relationships can be made explicit.

[Function-Value <Function Term> <Anything>] New
This predicate is true if Anything unifies with the value of the Function Term. A Function Term's value may only be set by Set-Function-Value. Note that this builtin is also available directly from Lisp.

[Genvalue <Variable / list> <Lisp-Expression>]
Sets the Rhet variable <Variable> to first value in list returned by evaluating the <Lisp-expression>. Other values are used for backtracking. If the Lisp expression returns more than one value (in the Common Lisp sense) then the Car of each of these values will be assigned to each variable in the list, with extras used for backtracking (*i.e.* successive Cadr). Extra variables are set to Nil. Extra values returned are ignored.

[Ground Term1]
Succeeds if Term is a fully grounded term, *i.e.*, it contains no variables.

[Identical Term1 Term2]
Succeeds if Term1 and Term2 are structurally identical, *i.e.*, if they unify without assignment of variables or the equality mechanism. For example, `[IDENTICAL A A]` succeeds, and `[IDENTICAL A ?x]` or `[IDENTICAL ?x ?y]` (both unbound) fails.

[Itype <Type Descriptor> *&Rest* <Term>] Changed
Sets the terms (any Rhet object) to be of the immediate type described. Note that this type is immutable. That is, given a type hierarchy that splits the type bird into flying-birds and non-flying birds, to have asserted an object to have the immediate type of bird makes it illegal to further classify it into flying or

non-flying via the type system. It is not legal to later change the type (via Utype or Itype) to a more specific type. It signals an error to set the type of an atom to be something that is not a subtype of it's existing type (see Utype), or to be anything other than it's immediate type should that have been declared. See also Utype, Dtype. This builtin is assertable.

[Member Term1 List**]**
Succeeds if Term1 unifies with a term on the List.

New

[Not <form>**]**
On the LHS of an axiom it indicates the axiom is used to disprove the enclosed horn clause (or nonenclosed predicate, i.e. `[NOT [P X]]` vs. `[NOT P X]`). On the RHS of an axiom, it indicates that the enclosed horn clause or unenclosed singular term is to be disproved using whatever proof mode is in force (*i.e.* complete if prove complete was engaged, or simple if simple or default reasoning modes are in force).

Expanded

[NotEQ? Term1 Term2**]**
Succeeds if Term1 and Term2 are provably unequal. That is, either their types would not unify, they are distinguished elements of the same type, or they have been asserted to be Add-InEQ. Otherwise it fails.

New

[Or <Form>*****]
Succeeds if any of the <Form>* can be proved. `[OR]` fails. `[Cut]` is not caught by this form, unlike And, so backtracking past a CUT in this form causes the rule in which it is used to fail. Note that Or will evaluate it's arguments specifically in the order supplied, and will short-circuit evaluation as needed. Thus the clause `[Or [P ?x] [Q ?x]]` will not cause evaluation of `[Q ?x]` if `[P ?x]` is true.

[Post <atomic formula>**]**
Succeeds if the atomic formula is ground and can be proven, fails if the atomic formula is ground and cannot be proven, and otherwise constrains all variables in the formula such that they can only be unified with objects that would have allowed the atomic formula to succeed. See section 7.5.2 on the Post Constraint Mechanism in the tutorial for a further description. Note that several other Rhet builtins use an implicit POST for certain arguments. See, *e.g.* DISTINCT.

[Relation-Form? Thing Relation Type] — New

Succeed if Thing is Equal to an element of the Relation list on Type, a structured type. If Thing is unbound, bind it to (a copy of) successive elements of the Relation-List for Relation on Type. Don't currently handle unbound vars for Relation or Type.

[Relation-List List Relation Type] — New

Succeed if List is Equal to the Relation list on Type, a structured type. If List is unbound, bind it to (a copy of) the Relation-List for Relation on Type. Don't currently handle unbound vars for Relation or Type.

[Retract Term1] — Current Status: Builtin unavailable.

Retracts all axioms whose head unifies with Term1. This builtin is also available from lisp.

[Rformat Stream Control-String *&Rest* Form*] — New

The values of the list Form* are printed according to the Control-String on Stream. This is just like the Common Lisp Format function. This builtin is assertable. Normally Stream would be :T for the default, but can also be a list that will evaluate to a stream. That is, this will translate into a call to (Apply #'Format (List* (Eval Stream) Control-String Form*)).

[Role Instance Role-Name Value] — Limited, Role-Name must be bound

Succeeds if the Rhet object Instance has role Role-Name with value Value. Value is a Rhet object, Role-Name is a Lisp Atom. See also Add-Role.

[Role? <Role Name Atom or Variable> <Type-Descriptor or Variable>] — New

Succeeds if the <Role Name Atom> is a Role defined or inherited by the <Type-Descriptor>. The Type Descriptor should be either an atom or a list, just like what would normally appear after the "*" on a typed variable (except for the colon needed to distinguish the term as an atom rather than a function term). For example, :T-Lisp would be used for type T-Lisp, and (T-A T-B) would be used for the intersection of type T-A and T-B. If the Role Name Atom position is an unbound Variable, the function will successively bind it to the roles of the Type-Descriptor. If the Type-Descriptor position is

an unbound variable, it will be successively bound to all declared types with the appropriate role defined. If they are both variables, they will map over all roles on all types with roles.

Expanded

[Rprint Term1 ...TermN]
The values of Term1 through TermN are pretty-printed on successive lines. This builtin is assertable.

[Rterpri]
Prints a line feed and even a carriage return. This builtin is assertable.

New

[Set-Function-Value <Function Term> <Anything>]
This changes the value of the Function Term to Anything. A Function Term's value may only be read by Function-Value. Note that this builtin is also available directly from Lisp.

New
Current Status:
Unimplemented

[Setall Var1 Form Var2]
Succeeds if Var1 is set to the list of all unique terms that satisfy Form[14]. For example, `[Setall ?x [P ?y ?z] ?z]` would set ?x to a list of all things that have been asserted as arguments to predicate P that make it true. For example, given `[P A B]`, `[P B C]`, and `[P D C]` ?x would be bound to ([B] [C]).

Expanded

[Setvalue <variable / list> <Lisp-Expression>]
Sets the Rhet variable <variable> to the value of the Lisp expression <Lisp-Expression>. Any logic variables in <Lisp-expression> are replaced by their logic bindings before Lisp evaluation. If the Lisp expression returns more than one value (in the Common Lisp sense), then the values are assigned to each variable in the list in turn. Extra variables in the list are bound to **Nil**. Too few variables cause the extra values to be discarded.

New

[Type? <atomic formula> <Type Descriptor or Variable>]
Succeeds if the atomic formula is of (or can be set to - if the atomic formula is a variable) the described type. If the type parameter is a variable, it is set to the "best" available type we have for the object.

[14] By this we mean that the variable Var2 is assigned to a ground term, such that the Form with Var2 so bound is found in the KB directly, as opposed to via backward chaining.

Normally this will be the immediate type, if it exists, otherwise, the most immediate type declared for atomic objects, or best fit for predicates. See also UTYPE. The Type Descriptor should be either an atom or a list, just like what would normally appear after the "*" on a typed variable (except for the colon needed to distinguish the term as an atom rather than a function term). For example, :T-Lisp would be used for type T-Lisp, and (T-A T-B) would be used for the intersection of type T-A and T-B.

Changed

[Unless <atomic formula>*]
Succeeds if any form in the formulas given cannot be proven, or can be proved as [NOT <atomic formula >)]. This gives us proof by failure. Note that variables change in interpretation in the UNLESS function, *e.g.*, if we are given the fact that [P A] is true, then

[UNLESS [P B]] will succeed,

[UNLESS [P A]] will fail as expected, but

[UNLESS [P ?x]] also fails, since [P ?x] can be proven.

New

[Unify <Term> <Term>]
Succeeds if the two terms unify without equality. This is faster than [EQ?]. It can also be used to determine when two terms are equal but distinct, as in

(7.47) [And [EQ? ?x ?y] [Not [Unify ?x ?y]]].

Was TYPE

[Utype <Type Descriptor> *&Rest* <Form>]
Sets the Form(s) (any Rhet object) to be of the type described. Note that this type is NOT defined to be the immediate type. It is legal to later change the type (via Utype or Itype) to a more specific type. It signals an error to set the type of an atom to be something that is not a subtype of it's existing type, or to be anything other than it's immediate type should that have been declared (see Itype). See also Dtype. This builtin is assertable.

[Var Variable]
Succeeds only if Variable is an unbound variable.

New **[Win]**
Always succeeds. This is mainly needed as a predicate inside of Cond forms.

New **[With UContext <Form>*]**
Similar to ASSUME, except that instead of building an arbitrary UContext, it creates a named UContext (if it didn't exist) and sets the proof UContext to that UContext. It then attempts to prove each of the Forms, succeeding in the same fashion that AND succeeds. UContext must be a string, the name of a UContext.

In addition a general modal operator for Belief is defined:

New **[XB <Atomic Formula>]**
This asserts the Atomic Formula, (if it appears in Assert-Axioms), or tests the Atomic Formula (if it appears in the RHS of a horn clause) into the belief space X. X can be any alphabetic character, except M, It may also appear on the LHS of a horn clause, with the usual semantics. This has the semantics described in section 4. These may be nested at will, for convenience, without parens, much as Lisp Cadr, that is

(7.48) `[SBHBTBQB[P] <]`

would translate to an assertion that relative to S's beliefs about H's beliefs about T's beliefs about Q's beliefs, P holds. Note that it is possible to move to the context set up by a belief operator, via the Operator Lisp function, but it's best to use the modal operator, whenever possible. This builtin is assertable.

New **[MB Agent1 Agent2 <Atomic Formula>]**
Like XB, but uses a special "Mutual Belief" space. To assert `[MB[S H P] <]` it is derivable that `[SB[P]]` and `[HB[P]]` as well as the full belief closure between these agents, (*e.g.* `[SBHBSBHBSB[P]]`). Restriction: the MB operator may not be used in an ASSUME clause. This builtin is assertable.

Note that all such belief statements are relative to the current DEFAULT UContext.

Chapter 8

Programmatic Interface

8.1 Manipulating Facts

8.1.1 Adding and Deleting Facts

Assert-Axioms *<Axiom1> ...<AxiomN>* &key *Context Justifications*

Adds the specified axioms to the data base. It returns a list of the axiom (or fact) accessors so added, thus succeeds when found on the RHS of a horn clause. For axioms, that is, if there is a RHS, one would normally use **DefRhetPred**, however. If the context parameter is specified, the context passed will indicate to the system which context axioms should be asserted relative to. Normally one would use the **Ucontext** and or **Operator** functions to convert a string into a context for **Assert-Axioms**. It will otherwise default to the default context, which the user can set programmatically[1]. The Justifications are added

Enhanced
Current Status:
Justifications not tested.

[1] It is the value of ***Default-Context***.

to any facts added via this form. Justifications are created by appropriately[2] massaging the proof tree from a BC proof. See also 3.4.

Retract *<Fact>* &Key *Context*

Retracts the (previously asserted) Fact[3] passed in the context Context. It is permissible to, for example, assert [Foo] in a context, and retract it in a child context. This would make [Foo] true in the asserted context, and all child contexts above the context it was retracted in. *I.e.* if we add context TC as a child of T, and TCC as a child of TC, and assert [Foo] in T and retract it in TCC, we would still see foo as true in contexts T and TC, but undefined in TCC[4].

Retractall *<Form>* &Key *Context*

Retracts all the facts in the database whose conclusions unify with the specified form. The predicate name must be specified in the form. If an atom is given as a pattern, it will be interpreted as a predicate name and all axioms for that predicate will be deleted. For example, (Retractall [P A ?x]) retracts all facts whose head unifies with [P A ?x] (*e.g.*, [P ?x ?z], [P ?x B], [P A B]), and (Retractall 'P) retracts all facts with head P[5].

Contrast to HORNE

Clear *RE Context*

Retracts all facts in the database with an index that could be generated by the regular expression. See 8.1.1.1, below, for more information on acceptable regular expressions.

---

[2]There should be a function, eventually, to do this, particularly since we haven't specified what is an appropriate munging.

[3]The function will also accept a fact-accessor, which happens to be the programmatic result of several of the fact accession functions

[4]In fact, we can assert [NOT Foo] in TC, and have Foo true in T, false in TC, and unknown in TCC.

[5]This is equivalent to specifying a form of [P &Rest ?x].

8.1.1.1 Regular Expressions

Rhet allows indexes, for commands that accept an index as an argument, to be a regular expression. Using the operators[6], "∧"[7] for "and" "∨"[8] for "or", "∈∈"[9] for the empty string, "[" and "]" to delimit subexpressions, and "*" for Kleene closure in a string, once can specify a miniature "program" to Rhet for finding indexes taken as a string. So, for example, the index "< ∧[*f* ∨ *b*] ∧ oo" would match fact indices of either "<foo" or "<boo".

8.1.2 Accessing Facts

Find-Facts *<Atomic Formula*>*

Returns all assertions of form [<conclusion>] or [<conclusion> <index>] that unify with the specified formula. Thus to find all facts that assert that P is true (or false) of something, we could use (Find-Facts [P ?x]). If the data base contained the facts [P A], [P B <a], [[P D] <b [Q R]], then the query would return[10](([[P B] <a] [[P A]]))[11].

Find-Facts-By-Index *'<Index>* &Key *Context* New

Returns all facts in the context whose Indices match Index. Given the data base above, (find-facts-by-index '<) would return[12] ([[P A] ¡]), while (find-facts-by-index "<a") would return ([[P B] ¡a]). Find-Facts-By-Index may be passed an Index either as an atom, or a string. An atom will search for exact matches to the atom, while a string may be any regular expression, used as described in section 8.1.1.1.

---

[6] A not operator and "+" should be added.

[7] Symbol-w on the Symbolics or Explorer keyboards

[8] Symbol-q

[9] Symbol-Shift-e

[10] Actually, it would return fact accessors for these terms, since they are more useful programatically.

[11] Find-Facts will *never* find axioms, only asserted true or false facts.

[12] As with Find-Facts, it would return fact accessors for these terms, since they are more useful programatically.

Find-Facts-With-Bindings *<Atomic Formula*>*
Same as find-facts except that it returns the variable bindings as well in the format[13] ((<axiom> <binding list>)*) for each form. For example, with the above three axioms for P, the query (find-facts-with-bindings [P ?x]) would return ((([[P B] <a] (?x [B])) ([[P A]] (?x [A])))).

New **Find-Fact-References** *<Form>**
Returns all facts[14] in the context that reference (has as an argument) all of the supplied Forms.

8.2 Manipulating Axioms

8.2.1 Adding and Deleting Axioms

Note that unlike Facts, Axioms may not be deleted from a particular context that is different from the context it was asserted in.

Assert-Axioms *<Axiom1> ...<AxiomN>* &key *Context*
See 8.1.1

Clear-Axioms
Delete all axioms defined by the user. Facts remain intact.

New **DefRhetPred** *Predicate-Name* (Argument-Lambda-List) &Body *Body*
Defines a Rhet predicate Predicate-Name that takes arguments according to the Argument-Lambda-List. Body is either a series of indicies and RHS definitions, or lisp code. These may not be mixed.

More specifically, the Argument-Lambda-List may be made up of the following:

[13] As with Find-Facts, it would return fact accessors for these terms, since they are more useful programatically.
[14] As with Find-Facts, it would return fact accessors for these terms, since they are more useful programatically.

1. & keywords, specifically:

   &Any the following variables (until the next & keyword) may be bound or unbound when the predicate is invoked, or may be bound to terms that are not fully ground. This is the default when no & keyword appears.

   &Bound the following variables are guarenteed by the programmer to be bound when the predicate is invoked. It is an error to attempt to prove the predicate without passing a fully ground term in this position. If (Declare (Optimize Safety)) appears, erroneous usage will signal an error[15]. If forms follow this keyword, variables embedded in the forms must be bound.

   &Forward this must be the last &Keyword specified. If present, the DefRhetPred defines FC axiom(s) rather than BC. The form(s) following this key are the trigger(s). If none are present, it is the same as having specified :All as the trigger. No unbound variables may appear in the trigger. The other part of the lambda list becomes a pattern for the fact to be asserted if the body of the DefRhetPred is proved[16].

   &Unbound the following variables are guarenteed by the programmer to be unbound when the predicate is invoked. If forms follow the keyword, variables within the forms must be unbound. It is an error to attempt to prove the predicate with any of the variables bound. If (Declare (Optimize Safety)) appears, erroneous usage will signal an error[17].

   &Rest the following variable must be of type T-List (it will be changed if it is not), and will be bound to all the remaining arguments of the predicate when attempting to prove it. This is just like the normal usage of &Rest in forms.

2. Rhet variables, which have the properties of the closest preceeding & keyword, as above, or &Any by default.

---

[15] Future functionality

[16] Currently, it is an error to have both &forward, and the body containing a lisp function.

[17] Future functionality

3. Rhet forms, which are pattern matched against the form being proved to see if this predicate is applicable.

The body then consists of the following:

1. If the first object is a string, it is considered a documentation string for the predicate.
2. If the first object (after the optional documentation string, if any) is a **Declare** form, it is taken to be declarations about the predicate, as for CL. All CL declarations are legal for predicates, if the embedded predicate definitions is a lisp function. If they are Rhet RHS terms instead, most such declarations are legal, but ignored. Other declarations that are legal include Foldable, Non-Foldable, Monotonic, and Non-Monotonic. See the definition for **Define-Subtype**, 8.12.1, for more information on these declarations, and how they interact with the constraints of the structured type subsystem.
3. After these optional declarations, if the next object is an index (*i.e.* it begins with the character "<"), then the following objects must all be legal Rhet terms or indicies. All terms until the next index or the end of the **DefRhetPred** will make up the RHS of an axiom (either BC or FC depending on whether the &Forward key was present in the Argument-Lambda-List), whose LHS is constructed from the Argument-Lambda-List.

   If another index is found, it begins the definition of a new BC/FC axiom, with the same LHS as defined by the Argument-Lambda-LIst.

   It is possible to have multiple **DefRhetPred** forms for the same predicate providing the Argument-Lambda-List can be distinguished between them, and all of them use this sort of body definition.
4. After the optional declarations, if the next object is not an index, the remainder to the body is taken to be an implicit Progn that defines an anonymous lisp function that will act as the predicate. At run-time, the lisp function will be evaluated, and if it returns non-nil, the predicate succeeds. Note that unlike Rhet predicate definitions in the previous item, only one **DefRhetPred** may be defined on a predicate if it is to expand into a lisp function. Note that if the lisp function will have embedded Rhet forms that contain references to the variables in the Argument-Lambda-List, the

DefRhetPred must be surrounded with the #[and #] special characters to put these references into a single environment, to assure they all refer to the "same" variable. See, for example, 7.19.

The idea behind DefRhetPred is to have a somewhat more concise representation (like DEFUN, DEFMACRO, these defining forms expand into the existing Rhet axiom definition forms). It is also a convenient place to put declarations that can be used by the compiler or the Rhet run-time/tracing system to help detect errors and improve efficiency[18]. Further, it is a simple mechanism for redefining groups of related axioms, similar to the incremental nature of DEFUN in lisp. (i.e. the new definition replaces the old, an addition axiom is not just added, leaving the old to be separatly deleted as using Assert-Axioms would require).

DefRhetPred claims to provide the complete definitions for a predicate. Other axioms or declarations made about the predicate are deleted on re-evaluation of the DefRhetPred form. This allows one to simply edit a particular portion of a DefRhetPred form and recompile it or reevaluate it to change the Rhet KB, rather than the much more laborious task of either making sure the indexes on the predicates are unique (so a particular one can be deleted), or having to delete all predicates with a particular name, and then reasserting all of them. Furthermore, the declaration style is very similar to lisp's (these are lisp forms, and not available as builtins) to make the syntax easy to remember.

For example:

(8.1)
```
(DEFRHETPRED P (&Bound ?x*T-U &Any ?y*T-List &Unbound ?z*T-List)
    ;; variables following a &Bound must always bound when the predicate
    ;; is run.  &Any indicates they may or may not be bound, and &Unbound
    ;; indicates they will NEVER be bound.  &context may also appear to *set*
    ;; the context the predicate is operative in.  Similarly for &forward to
    ;; define a trigger.  &forward w/o a trigger is an implicit :all.
```

[18] While currently most such declarations will be ignored, it seems like a good idea to have developed Rhet code get into the habit of using such declarations, if for no other reason than documentation.

```
(DECLARE (OPTIMIZE SPEED SAFETY) ; pass on to compiler
         MONOTONIC)              ; monotonic declaration means once
                                 ; asserted, it will never be retracted.
                                 ; (it is an error if it is)

;; if the first thing here is an index, we are defining the predicate
;; as a series of Rhet rules.

<1 [FOO ?X ?Z ?Y]
   [BAR ?Y]

;; note that an index separates the rules.

<2 [MUMBLE ?Z ?X ?Y])
```

(8.2)

```
(DEFRHETPRED Q (&LOCAL ?X*T-HUMAN ?Y*T-ATOM)
   (DECLARE (OPTIMIZE SPEED)
            NON-MONOTONIC)

   ;; First thing here is not an index, so we are defining a LISP function
   ;; as a predicate.  This will create an anonymous function and the lisp
   ;; form passed to the compiler.

   (IF (EQUAL ?Y :TEST)
       (PROVE [MUMBLE ?X])
       (ASSERT-Axoms [MUMBLE ?X])))
```

Note that the above functions should automatically expand into the appropriate lower level Assert-Axioms, as well as internal functions for the declarations that are not currently documented, which will not otherwise be available.

Given the above, if I were to eval:

(8.3)
```
(DEFRHETPRED Q (&local ?x ?y)
    ...)
```

I would NOT redefine Q[19], because the predicates are sensitive to their argument lists (note that ?x was not specified to be of type *T-HUMAN).

If I wanted to CHANGE the definition of Q's argument list, rather than specify an alternative LHS for predicate Q, I could do:

(UNDefRhetPred Q (&local ?x*T-Human ?y))

which would undo the defrhetpred. Again, this function is sensitive to argument lists.

Remove-B-Axioms *<Axiom LHS>* Was Retractall

All backward chaining axioms that have a LHS which will unify with the one passed are removed from the KB.

Remove-B-Axioms-By-Index *Index* New

Taking the Index as a regular expression, all BC axioms whose Index match, are removed. This index may be either an atom or a string; see 8.1.1.1.

Remove-F-Axioms *Trigger* Was Retractall

Removes all forward chaining axioms with a trigger that unifies with the passed Trigger.

[19] If the DefRhetPred is for a lisp function, then it is not sensitive and would cause a redeclaration.

New

Remove-F-Axioms-By-Index *Index*
As for BC axioms.

Was Reset

Reset-Rhetorical
Restores system to original unused state. This is loosely the equivalent of cold booting an empty world and loading the system from scratch.

New
Unimplimented

UnDefRhetPred *Lambda-List*
Undefines a previous DefRhetPred, with matching Arglist. If the DefRhetPred was for a lispfn, it is undefined regardless of arglist matching.

8.2.2 Examining Axioms

Changed, was axioms-by-name-and-index

List-F-Axioms *Trigger*
Returns a list of all axioms that match the literal trigger. A trigger of [X :all] will return all axioms with trigger head X.

Changed, was axioms-by-name-and-index

List-B-Axioms *<Axiom LHS>*
All BC axioms whose LHS matches the one passed are returned. A parameter of [X :all] will return all axioms with LHS head X.

Changed, was axioms-by-index

List-F-Axioms-By-Index *Index*
As usual, the Index may be either an atom or a regular expression as per section 8.1.1.1. All FC axioms that match are returned.

Changed, was axioms-by-index

List-B-Axioms-By-Index *Index*
Similar to the above, but for BC axioms.

8.3 Proof We Must

Prove *<Atomic Formula1> ...<Atomic Formulan>* &Key *Proof-Mode Context*

Attempts to prove the list of formulas, and returns a bound solution if one is found. This will be a list of the Atomic Formulas in the same form as given to Prove. By default Prove will use the proof mode currently in force (see section 10) which is initially default reasoning mode. The user may specify :Proof-Mode :Default, :Proof-Mode :Complete, or :Proof-Mode :Simple to force the issue. Note that as in 7.18, if the forms passed to Prove-All contain variables that are the same between these forms (Rhet forms at top level normally use distinct variables, even if they "print" the same), then the #[and #] characters should be wrapped around the list of the goals to be proven.

Prove-All *<Atomic Formula1> ...<Atomic Formulan>* &Key *Number-of-Proofs Proof-Mode Context*

Does an entire search of the axioms and returns all solutions found. If Number-of-Proofs is given a value of some integer, only the first integer proofs are returned. The value of Proof-Mode is set as in the Prove function. Note that currently if there is an infinite path in the proof tree (*e.g.*, a transitivity axiom) then this function will not return, unless bounded with the :Number-of-Proofs keyword. Will return a list of the lists of the formulas with their variables bound appropriately, *e.g.*, (Assert-Axioms [[happy joe]<] [[happy mary]<] [[sad frank]<]), (Prove-All [happy ?x] [sad ?y]) returns (([happy joe] [sad frank]) ([happy mary] [sad frank])). Note that as in 7.18, if the forms passed to Prove-All contain variables that are the same between these forms (Rhet forms at top level normally use distinct variables, even if they "print" the same), then the #[and #] characters should be wrapped around the list of the goals to be proven.

8.4 Now where did I put that?

Rhet allows the creation of a hierarchy of User Contexts, or UContexts. All functions that specify that they take a context as an argument, actually want RNContexts, as produced by some of these functions[20]. The distinction between a UContext and an RNContext is that the latter is an explicit Rhet Normal Context, which is a combination of a UContext and an implicit or explicit modal operator, e.g. "MB", "SB", or "HBMB". We start with the default Context "T", and if we add something, it goes into this context. But it's RNContext is "SBMB-T" because "SBMB" is the default modal operator. Other combinations of modals and UContexts are possible, what is important to know is that RNContexts are only created when they are needed, since the number of modals that may apply to a context are potentially infinite. To create a new UContexts, say "Foo", is to (implicitly) create a child context for each of the leaf RNContexts. Thus "SB-T" would be the parent of "SB-Foo"[21].

New **Convert-Name-to-UContext** *Name*

Given a Name, find the corresponding UContext, if there is one, otherwise it enters the debugger to force the user to supply the name of an existing UContext. If this error processing is unnecessary, or undesired, use Ucontext-P.

New **Create-UContext** *Name Parent-Name*

Make a new UContext with name Name and parent Parent-Name. It will be initially devoid of children and RNContexts.

---

[20] They are described here as RNContexts rather than as Contexts, because a Ucontext is actually a mapping of a user's string to a structure, which can be mapped to one of several RNContexts. There may be Contexts created directly using the internal mechanisms that are not RNContexts, but still allowed. This would be for the programmer who was using Rhet as a programming language, for instance, and wanted a more flexible view of Contexts.

[21] Note that "MB" Contexts are *not* created by creating a new UContext, since the intuitive construction would not give us tree-like context inheritance, or change the existing context inheritance, which Rhet does not allow.

Destroy-UContext *name* New

Forcibly remove a UContext and all its descendants, including all contained RNContexts.

Operator *Name* &Key *Ucontext* New

Given the name of a modal operator, Operator, tries to determine an associated RNContext (meant to be used as the argument to the :Context keyword). Normally the Ucontext will be the default one, but the user may specify a particular Ucontext, with the keyword. That is, to make the default context be "Foo" and the default operator "SBHB", one would do:

(8.4) `(Setf *Default-Context* (Operator "SBHB" :Ucontext (Convert-Name-to-UContext "Foo")))`

UContext *Name* New

Given the name of a UContext, tries to determine an associated RNContext (meant to be used as the argument to the :Context keyword). Note that the default modal operator in use at the time will be used in determining the RNContext. If you want to specify an operator as well, use the Operator function, described above.

UContexts New

Return a tree of all the UContexts known to the system[22].

UContext-P *Name* New

Is Name a UContext? Returns the UContext if it is, else Nil. If you would like to assume that you will always get a valid UContext returned, use Convert-Name-to-UContext, instead.

8.5 Unity

Rhet allows the user to call the unifier directly.

[22] For a list of all RNContexts known to the system, including some internal ones used by Rhet, call (Hname:Contexts).

New

Unify *<Atomic Formula1> <Atomic Formula2>*

This returns several values: the first is a success indication, and is non-**Nil** if unification between the two formulas succeeds. The succeeding values are the Rhet variables as they were bound by the unification (Rhet destructively modifies the passed variables to indicate their bindings). **Unify** ignores equality assertions.

New

E-Unify *<Atomic Formula1> <Atomic Formula2>*

Similar to **Unify**, but this call does E-unification, which Rhet normally does by default during proofs.

8.6 Consing Forms

Normally, if one has set one's readtable appropriately[23], one can freely embed Rhet forms within lisp expressions. Given a useable implementation of Common Lisp, these will self-evaluate, and not need to be quoted. It may, however, be desirable to CONS Rhet forms on the fly. Currently[24], one must do this by CONSing up a string and then doing a Read-From-String on the resultant object, *e.g.* as in the following example:

(8.5)
```
(DEFUN CREATE-INSTANCE (TYPE &REST ROLE-VALUE-PAIRS)
    "Like Define-instance, but returns the instance created, instead of being supplied it."
    (LET ((INSTANCE
            ;; kluge city
            (READ-FROM-STRING (CONCATENATE 'STRING "[" (STRING (GENSYM "foo")) "]"))))
      (APPLY #'DEFINE-INSTANCE (LIST* INSTANCE TYPE ROLE-VALUE-PAIRS))))
```

[23] E.g. by setting the Syntax attribute to "Rhet", as per 9.2.3.2

[24] See Enhancement Schedule, B.2

8.7 The Rhet/Lisp Interface

So far, we have seen how the various Rhet facilities can be invoked from within Lisp. This section explains how Lisp facilities can be used within Rhet.

8.7.1 Calling a Lisp predicate directly

It is possible to make the proof of some Rhet form directly depend on a Lisp predicate. This is done with the following function:

[Call <Lisp Expression>] New

This predicate simply invokes the Lisp expression, and if it returns non-Nil, it succeeds. For more detail, see 7.44. Note that the Lisp expression is expected to be a list, not a function, so Rhet may replace variables mentioned with their bindings (this means the user function doesn't need to know how to handle bound variables).

8.7.2 Assigning Lisp Values to Rhet Variables

There is a simple mechanism for binding a Rhet variable to an arbitrary Lisp value. This is accomplished by using the built-in predicate:

Setvalue *<Variable-List> <Lisp expression>*

This evaluates the <Lisp expression> as a Lisp program and binds the results to the Rhet variables specified. If the variable is already bound, SETVALUE will fail. If the Lisp expression returns multiple values, each value is bound to each variable in the variable list in turn. Extra variables specified are bound to NIL. Extra values

returned are ignored. Note that this is also a Rhet predicate, and may appear inside of axioms[25]. For example, the common Lisp function Floor returns two values, the integer that is the floor, and the remainder. Calling

(8.6) (Setvalue '(?x ?y) '(floor 13.5))

would bind ?x to 13 and ?y to 0.5.

Genvalue <*Variable-List*> <*Lisp expression*>

This is the same as [Setvalue] except that the Lisp expression is expected to return a list of values. The variable will be bound to the first value, and if the proof backtracks to this point, to the succeeding values one at a time. If the Lisp expression returns multiple values, each value is assumed to be a list, and each value returned is associated with a variable in the variable-list. Note that this is also a Rhet predicate, and may appear inside of axioms. A slightly modified example 8.6 would give us something like:

(8.7) (Genvalue '(?x ?y)
'(Values (Multiple-Values-List (floor 13.5)) (Multiple-Values-List (floor 27.82))))

which, after parsing through the Lisp multiple values, would bind ?x to 13 and ?y to 27, then if backtracking happened, would next bind ?x to .5 and ?y to .82.

8.7.3 Lisp Functions as Predicate Names

Occasionally it is useful to let a predicate name be a Lisp function that gets called instead of letting Rhet prove the formula as usual. These special Lisp functions will be applied to a frozen form of the arguments coded, rather that to the actual rhet forms. They cannot support backtracking or side effects, so the need for either should be avoided. They receive their argument list from Rhet with all bound variables replaced by their values. To declare such a Lisp function to Rhet use:

[25] It is, frankly, much more useful as a predicate, however, it does allow user functions called by Rhet via Call or Declare-Lispfn to handle unbound variables.

Declare-Lispfn <*Name*> *Query-Function-Symbol* &Optional *Assert-Function-Symbol Type-Declarations*

Declares to Rhet that Name is not a predicate but a Lisp function; Rhet will recognize those <Name>s as calls to Lisp functions. If the Reasoner is attempting to prove an axiom that has been declared a Lisp function, it will call the Query-Function-Symbol (passed as a symbol to allow it to be incrementally recompiled). If it attempting to add a predicate to the KB whose head is declared with an associated Assert-Function, it will call the Assert-Function-Symbol rather than add it[26]. Lisp Query-Functions should only return "t" or "nil" which will be interpreted as true and false respectively. The optional Type-Declarations are a list of symbols representing the types of the arguments expected by the lispfn. This will be used for runtime debugging interaction, and as a hint to the compiler. See also [Miller, 1989] for information on lisp functions that are available to deal with rhet objects that may be handed as arguments to a lispfn.

For example, assume we enter the following:

```
(defun check (&rest x)
    (terpri)
    (princ "in check, args are:")
    (princ (Mapcar #'ui:real-rhet-object x))
    t)
```

(8.8) `(Declare-Lispfn 'check 'check)`

(8.9) `(Assert-Axioms [[P ?x ?y] < [check ?x ?y]])`

Then if we call

(8.10) `(Prove [P A B])`

[26] It will still forward chain as if it had added it, although it may not detect a loop!

the Lisp function check is called resulting in the output:

`in check, args are:` (A B).

Since check returns a non nil answer, the Lisp call is treated as a success.

Note that the package of the assert and predicate symbols are significant, but the package of the Name of the lispfn is not. So, there may only be one lispfn by a particular name; Rhet does not support pacakges within forms.

Lispfns cannot backtrack, and therefore cannot bind variables. Neither can they have side-effects you want Rhet to be able to undo. If you need either of these things, you need to define your own builtin. See [Miller, 1989], the Rhet programming guide, for more details on how to go about doing this. If a lispfn is called or asserted with an unbound variable embedded in it's argument list, Rhet will signal an error.

A few useful functions for manipulating argument lists within Lisp are:

New **Hname:Rvariable-P** *<Any Lisp Object>*
Returns NIL if the object is not a rhet variable structure.

New **Hname:Rvariable-Pretty-Name** *<Variable>*
Returns the variable name.

New **Hname:Rvariable-Type** *<Variable>*
Returns the type of the Rhet variable.

Note that both of the above functions expect to operate on actual variable structures, as Rhet would give to a unsuspecting lisp function if a variable is unbound, or as the reader would produce as the value of typing "?x". Other useful functions can be found in the Programmer's Guide.

8.7.4 Using Lists in Rhet

Rhet is embedded in Lisp, and one can use the Lisp list facility directly. The Rhet unifier normally operates on an internal structure called a Form. This is what an expression such as [Father-of Mary] is turned into by the reader. But unification will work on Forms or Lists. Lists, however, do not unify with Forms; they only unify with other Lists. Further, few builtins will work with Lists, normally Forms are expected as arguments.

The unifier will handle the dot operator appropriately anywhere it is syntactically legal. Thus the following pairs of terms unify with the most general unifier shown:

(a b c) (a ?x ?y)	with m.g.u. ?x/b, ?y/c
(a b c) (a . ?x)	with m.g.u. ?x/(b c)
(a b c) (?x . ?y)	with m.g.u. ?x/a, ?y/(b c)
(a b c) (a ?x . ?y)	with m.g.u. ?x/b, ?y/(c)
(a b) (a ?x . ?y)	with m.g.u. ?x/b ?y/nil
(a) (a ?x . ?y)	does not unify.
(a b) (?x)	does not unify. (?x) only matches lists of length 1.

Form unification is also allowed with varying arity predicates.[27] The main difference is that rather than the Lisp dot operator, the lambda form &Rest is used instead. &Rest binds all the remaining parameters of a function term to the Rhet variable following the declaration as a List of Forms. It is semantically equivalent to a dot operator, but that is only legal in Lists and not Forms. This List can be decomposed in the usual manner, as described above[28]. New

Consider the definition of the predicate [or*] that is true if any of its arguments is true[29]:

[27] Even with a var in the predicate name position!

[28] I will note again that by default the type of a variable is T-U, when it is present in a form, which cannot be bound to a List. Therefore a form such as [?x &rest ?y] will never unify with anything. The user should instead use [?x &rest ?y*T-LIST] or [?x &rest ?y*T-Lisp]. The default for a variable inside of a list is T-Lisp.

[29] Actually, the builtin Or would accomplish this, but for illustrative purposes...

```
[[or* ?x &rest ?y*T-List] < ?x] ; or* is true if the first argument is true
[[or* ?x &rest ?y*T-List] < [or** ?y]] ; or* is true if or* of all but the
                                        ;first argument is true. Note or** usage.
[[or** (?x*T-U . ?y)] < [or* ?x]]       ; need or** to handle LIST vs. FORM syntax
[[or** (?x*T-U . ?y)] < [or** ?y]] ; note typing on variables.
```

Thus the call with no arguments, `[or* ]`, always fails and each of `[or* [A]]`, `[or* [B] [A]]`, and `[or* [B] [A] [C]]` succeeds if `[A]` is provable. The clumsiness of the two different representations[30] for LISTs and FORMs is what drives us to have to write both or* and or**; this is only a problem for varying arity predicates (those that use &Rest).

Note, again[31], that there are two direct interfaces to the unifier from Lisp:

New

E-Unify *Term-or-Form Term-or-Form* &Key *Context*
Uses the equality subsystem while resolving the unification. This is what Rhet normally will use during a proof. Returns a success indication and a list of the variables bound.

New

Unify *Term-or-Form Term-or-Form* &Key *Context*
Like E-Unify, but disables the equality subsystem during the unification.

A quick example: Lets say we know that `[A]`, `[P Q]` and `[B]` are all declared equal.

- `[A ?x]` and `[B C]` will E-Unify with m.g.u. ?x/`[C]` but will not Unify.
- `[A &rest ?y*T-List]` and `[A B C D]` will both Unify and E-Unify with m.g.u. ?y/(`[B]` `[C]` `[D]`).

[30] Horne had only one, but lost expressivity

[31] See 8.5 for a more complete function description of Unify and E-Unify.

- [P ?x] and [B] will not Unify, but will E-Unify with m.g.u. ?x/[Any ?x [EQ? [P ?x] B]]. This is intuitive when you remember that there may be more than one binding of ?x that satisfies the equality, and we would not want to prematurely bind it.

- [P ?x] and [C] would neither Unify nor E-Unify. It will not E-Unify because there is *currently* no binding of ?x possible that could make [P ?x] equal to [C].

8.7.5 Manipulating Answers from Rhet

Since Rhet deals with it's own ideas of how variable values are found, *etc.* some Lisp functions are included to allow user code to more easily manipulate the results from certain functions.

Get-Var-Binding <*Variable-Name*> *Bindings* — Note difference.
Returns the binding for the named variable. For example,

(8.11) (getbinding '?x '((?x FOO) (?y BAR)))

will return FOO. Normally this would be called on the result of Prove.

Unify:Get-Binding <*Variable*> — New
Returns the binding for the variable. This is an actual Variable Structure, as is returned by the reader on "?x" or can be passed by Rhet to Lisp functions.

List-Forward-Chained-Facts — New
This function returns a list of all axioms that have been asserted via forward chaining since the last call to this function.

8.8 Equality

This subject is discussed in further detail in section 3.2.

The horn predicate Add-EQ is also available as a Lisp function:

Was EQ

Add-EQ *Term1 Term2* &Key *Context*
Both terms must be fully grounded. This Lisp form will add the equality specified. Note that the context the equality is added in may be explicitly specified. Note that this function is also available as a Rhet predicate.

There are three Lisp functions for examining the equality assertions:

Equivclass <*Ground Term*> &Key *Context*
Returns a list of all ground terms[32] equal to the <Ground Term> in the provided Context.

Equivclass-V *Term* &Key *Context*
Returns a list of all terms[33] that could be equal to the term followed by variable binding information, in the provided Context.

New

Primary <*Ground Term*> &Key (Context *DEFAULT-CONTEXT*)
Returns the fact struture of the primary instance of the class the passed term is a member of. This is usually the simplest member[34] of the class, *e.g.* if we know

(8.12) `[Add-EQ [Mother-of John] [Francene]]`

then

[32] As with Find-Facts, it would return fact accessors for these terms, since they are more useful programatically.
[33] As with Equivclass, it would return fact accessors for these terms, since they are more useful programatically.
[34] That is, the function term with the least number of arguments.

(8.13) (Primary [Mother-of John])

will return [Francene]. Note that the system decides when it is approprite to update the primary object for a canonical class, it is not something the user can declare.

8.9 Inequality

The horn predicate Add-InEQ is also available as a Lisp function:

Add-InEQ *Term1 Term2* &Key *Context* New

Both terms must be fully grounded. This Lisp form will add the inequality specified. Note that the context the inequality is added in may be explicitly specified. Note that this function is also available as a Rhet predicate.

There is also a Lisp function for examining inequalities.

InEquivclass <*Ground Term*> &Key *Context* New

Returns a list of all ground terms[35] that could not be made equal to the <Ground Term> in the provided Context. This may be because the terms are of disjoint types, distinguished subtypes of the same type, or declared to be unequal via Add-InEQ. Note that this is an expensive operator and should only be used in exception circumstances.

[35] As with Find-Facts, it would return fact accessors for these terms, since they are more useful programatically.

8.10 Function Term Values

Function Terms can not only be set Equal or Inequal to one another, but they may also have Values. Only the following functions are able to manipulate or examine the value of a Function Term.

New

Function-Value *<Function Term>* &Key *Context*
Return the value cell of a Function Term. Note that this function is also available as a builtin (as a predicate).

New

Set-Function-Value *<Function Term> Value* &Key *Context*
Set the value of the Function Term to Value. Note that this function is also available as a builtin.

See 5.2 for an example.

8.11 Types

8.11.1 Adding Type Information

The following Lisp functions are supported for adding type information... Note that Rhet does not support type manipulation during proofs, so these functions are not available as builtins.

Tsubtype *Supertype Subtype** — Was axiom Isubtype

Asserts that each type in Subtype* is a subclass of the Supertype, *e.g.*, (TSUBTYPE 'ANIMAL 'CAT) asserts that CAT is a subclass of ANIMAL.

Tdisjoint *TypeName** — Was axiom Disjoint

Asserts that all the types mentioned are pairwise disjoint.

Tname-Intersect *Newtype TypeName** — Was axiom Intersection

Asserts that the intersection of all TypeName* is Newtype. *E.g.*, (Tname-Intersect 'Woman 'Human 'Female) says that objects that are both subtypes of type Human and type Female are of type Woman. Note that if more than two types are present in TypeName*, Tname-Intersect may create new named intersections in order to appropriately build the type table. Warnings are given when this happens.

Toverlap *TypeName** — New

Asserts that all TypeName* overlap, but that their intersection is unnamed. This is not needed when type assumption mode is active, since all types not known to be disjoint will be assumed to overlap. See 10.2.

Txsubtype *Super-Type Type** — Was axiom Xsubtype

Asserts that all type* are a partition of Super-Type, *i.e.*, they are all subtypes of Super-Type, that all Type* are pairwise disjoint, and that the union of all Type* is equivalent to Super-Type[36].

[36] Currently, at least, Rhet only recognizes Txsubtype as an abbreviation for Tsubtype and pairwise Tdisjoint declarations. Eventually, we hope to be able to take advantage of the additional information Txsubtype implies.

The system that adds a TYPE declaration and its implications to the matrix first checks that the statement is consistent. If the statement contains an inconsistency, an error message is printed and no information is added to the matrix. For example, if one adds (Tdisjoint 'cats 'dogs) and then adds (Tsubtype 'cats 'dogs), an error message will be given and information in the second declaration will not be added to the matrix.

In order for the matrix system to derive all implied information, Itype assertions should be added after Tsubtype, Txsubtype, Tdisjoint, and Tintersection assertions. In fact, the entire type table should be in place before any Assert-Axioms are done. The type table is expected to remain constant during a proof, therefore declaring any of the above to be Lisp functions to Rhet, and calling them from inside of an axiom is not recommended.

Type restrictions on the arguments to a function term, and on the type of the function term itself, are declared using the form:

Slightly different

Declare-FN-Type *Fn-Atom* (Type1 ...TypeN TypeN+1) &rest *Function-Specs*

Asserts that Fn-Atom is the name of a function that takes arguments of the types Type1,...,TypeN and describes objects of type TypeN+1. For example, (declare-fn-type 'ADD '(Odd Odd Even)) declares a two-place function ADD, such that when both arguments are of type Odd, it will produce an object of type Even. Other examples:

- (Declare-FN-Type 'Father '(Human Man))
- (Declare-FN-Type 'Spouse '(Human Human) '(Man Woman) '(Woman Man))
- (Declare-FN-Type 'Parent '(Human Human))

Function-Specs is an arbitrary number of *Function-Spec-Clauses*, where each Function-Spec-Clause has the form (Type1 Type2 ...Type n+1) In general we use 'function specification' to refer to typing information about function terms. Informally this means that if the i-th argument is of type-i (i = 1,...,n) then the result is of type-n+1. If a list of functions appears as the first argument as in the third example above, the same specification is declared for all those functions. If there was an old declaration for function-atom, it will be overridden. Declare-FN-Type returns all the arguments as they are given.

Type specification is actually associated with p-names (string) of function-atoms thus it doesn't matter in which context a user makes a declaration and a declaration is effective for all the contexts.

Note that the macro DefRhetFun is a somewhat nicer interface to this function, and is recommended to be used instead of Declare-FN-Type directly from the top level.

The declarations to Declare-FN-Type will also be inverted. That is, given the above example for Spouse, if a term [Spouse ?x] is found and known to be of type Woman, ?x will be derived to be of type Man. Had we instead declared

(Declare-FN-Type 'Spouse '(Human Human) '(Male Female) '(Female Male))

, which seems a reasonable declaration, we run into two problems:

1. This reverse typing cannot be done, because no result is of type Woman. While in theory we could derive such information, the current implementation does not do so.
2. When we specify types for Declare-FN-Type we must assure that it works for all subtypes, and the types are contained in a hierarchy. That is, in this restatement, Male is not a subtype of Human, nor is Human a subtype of Male. It is an error to give Declare-FN-Type arguments that are not specializations of the first argument. This example could be "fixed" by making the first declaration (T-U T-U), but in general we may not want to do this so we can trap bad arguments to the Spouse function, *e.g.* an argument type that is not a Human.

So, the rules of thumb are:

1. The most general argument type should come first, and all arguments to the typed function must be subsets of this type.[37]

[37] Rhet will give a warning (upon use) if this rule is violated; Rhet is free to coerce an axiom or term of the form [[P ?x ?y] < [EQ? [Spouse ?x] ?y]] into one where ?x and ?y are both typed Human (given the example definition for Spouse) because the declaration said that Human was the most general type Spouse would accept, not T-U.

2. Until Rhet is smarter, there should be a result form for each possible result type you expect the function to return so inversion works.
3. Each declaration must be independantly true. That is, again for Spouse, we have said that if the argument is of type Human, *or a subtype of human*, then the result will be of type human *or a subtype*. Thus the following would be an error:

(8.14)
```
(Declare-FN-Type 'Foo
                 '(Human Human T-Lisp)
                 '(Human Man T-List)
                 '(Man Human T-List)
                 '(Man Man T-Atom))
```

because the last declaration is in conflict with the second and third; if both arguments are of type Man, then the result is of type T-List by both the second and third entry, but of the *disjoint* type T-Atom by the last. In general, the more specific the arguments, the more specific the result type should be, and always a subtype of the less general ones.

A user can access/modify function specifications using the following functions. The syntax of each function is exactly like declare-fn-type — each allows one or a list of function atoms as argument (except 'look-up-fn-type' which takes one function-atom only) and none of the arguments is evaluated so a user doesn't have to quote any argument. They pretty much do what you expect them to do and all of the modification functions (*i.e.* all except Look-Up-FN-Type) return a list of arguments as given.

Was Declare-fn-type

Add-FN-Type *Function-Atom/List-Of-Function-Atoms* &Rest *Function-Spec*
Function-Spec is added to (each) Function-Atom.

New

Clear-All-FN-Type
Removes all the function specifications declared so far.

Delete-FN-Type *Function-Atom/List-Of-Function-Atoms* &Rest *Function-Spec*
Each clause appearing in Function-Spec is deleted from the specification of (each) Function-Atom if it exists.

Get-All-FN-Type — Was Defined-functions
Returns all the function specifications declared so far.

Look-Up-FN-Type *Function-Atom* — Was see-function-definition
Returns a function specification for function-atom. Returns Nil if none has been declared.

Remove-FN-Type-Def &Rest *Function-Atoms* — Was delete-fn-definition
Removes the function specification associated with (each) function-atom.

DefRhetFun
Function-Name ((Most-General-Arg-Type1...Most-General-Arg-TypeN) Most-General-Result-Type)&Body Body — New Unimplimented
A macro to make top level definition of function terms more painless. It claims to provide the complete definitions for a function term. Other declarations made about the predicate are deleted on re-evaluation of the DefRhetFun form. This allows one to simply edit a particular portion of a DefRhetFun form and recompile it or reevaluate it to change the declarations, rather than the more laborious task of using Remove-FN-Type-Def, then using Declare-FN-Type to redefine it. The declaration style is intensionally very similar to lisp's (these are lisp forms, and not available as builtins) to make the syntax easy to remember.

The body may consist of the following:

1. If the first object is a string, it is considered a documentation string.
2. If the first object after the optional documentation string is a Declare form, it is treated like CL Declare forms, that is, as instructions to the implementation that may assist in generation code or debugging information, but not change the language construct itself (the correctness of the program and the results that are derivable from a correct program are independant of declarations).
3. After either of these optional constructions, a keyword is expected. This may be one of the following:

:Alternates which is followed by one or more alternative type definitions, as accepted by Declare-FN-Type. These will be evaluated, so should be quoted if necessary. All of the types must be subtypes of the respective Most-General-Arg-Type or Most-General-Result-Type, and follow all the other rules of typing that Declare-FN-Type requires.

:Function-Cell-Default which is followed by a single evaluated lisp expression that will set the function cell of any occurance of this function. Since the evaluation is done at run time, it is permitted to check the type of the object, if necessary, if it depends on the type of the argument, but remember that the function will be run the first time the function is seen, not necessarily when embedded variables are typed and bound. The variable ?self is special in this context, and is bound to the function instance whose cell we are updating. See also Set-Function-Value, which has the same effect, but at runtime; the :Initializations option on Define-Subtype and Define-Functional-Subtype provides similar functionality, but for specific types, rather than specific functions.

An example:

(8.15)
```
(DEFRHETFUN MOTHER-OF ((T-SEXED-ANIMAL) T-FEMALE)
    ;; most general allowed types first.  If safety is on, and Father-of is
    ;; used on an argument not a subtype of T-Sexed-Animal, we get an error.
    (DECLARE (OPTIMIZE SAFETY))

    :ALTERNATES ; alternative types
    '((T-HUMAN) T-WOMAN)
    '((T-DOG) T-BITCH)
    ;; what we'd REALLY like (but can't yet express) is something like:
    '((?X*T-SEXED-ANIMAL)
      [ANY ?Y*FEMALE (TYPE-COMPATIBLEP (GET-TYPE-OBJECT ?Y)
                                       (GET-TYPE-OBJECT ?X))])
```

```
:FUNCTION-CELL-DEFAULT
;; set a default value on the function cell of this function
(FIND-MOTHER (HNAME::CACC-PRIMARY ?SELF))) ; what the hell.
```

Note that the above functions should automatically expand into the appropriate lower level DECLARE-FN-TYPEs, as well as internal functions for the declarations that are not currently documented, which will not otherwise be available.

Given the above, if I were to eval:

(DEFRHETFUN MOTHER-OF ((T-SEXED-ANIMAL) T-FEMALE)
...)

I would redefine the mother-of function, independant of the arglist I used.

Examples and further discussion on function typing are found in the system overview, Section 3.1.1.

8.11.2 Deleting Type Axioms

Note that Rhet types can only be deleted via the enhanced user interface[38]. Use Reset-Rhetorical.

8.11.3 Lisp Interface to Type System

There is a set of Lisp functions to access and use the type system independently of Rhet. The most important function returns the type of an arbitrary Rhet term:

[38] And right now, even that isn't possible!

Current Status:
Unimplemented.

Get-Type-Object *Term*

Given any Rhet term, this function returns the most specific type of that term. If the term contains one or more variables, it returns the most specific type that includes every instantiation of the term. This may or may not depend on a functional type for the object being defined.

New

Type-Compatiblep *Type1 Type2*

Takes any two types and returns T if the types are identical, or if Type1 is a proper subtype of Type2[39].

New

Type-Supertype *Type* &Key *Recursive*

This returns the immediate supertypes of the type as defined, if known. If the Recursive option is supplied and non-nil, this function returns the closure of all immediate supertypes.

New

Type-Subtype *Type* &Key *Recursive*

Returns a list of the immediate subtypes of a type. If the Recursive option is supplied and non-nil, this function returns the closure of all immediate subtypes.

New
Current Status:
Complex types are not supported.

Type-EQ *Type1 Type2*

Returns T if Type1 is equivalent to Type2. For example, if we have defined types B and C as exclusively partitioning type A, then we would expect (Type-EQ 'B '(A-C)) to succeed. The type reasoning supported by the type subsystem may not be general enough to derive that two types are Type-EQ even though they are. For example, if several partitions of a type are defined, and other constraints on these partitions are known, the type subsystem may not completely derive other relationships between these partition types.

New
Current Status:
Unimplemented.

Type-EQL *Type1 Type2*

Returns T if Type1 and Type2 overlap with no named elements NOT in common, *i.e.* $Type1 \cap Type2 = Type1$. By named element, we mean type, *e.g.* if T1 and T2 are defined to overlap, but T3 is a subtype of T1 and not of T2, then T1 and T2 are not Type-EQL.

---

[39]This is essentially a nice user interface to the internal function Typecompat, which is one of the primary interfaces between Rhet's unifier and the type subsystem.

Type-Subtypep *Type1 Type2*
Succeeds if $Type1 \subset Type2$. This is distinct from Type-Compatiblep, in that we must find some defined or inferred type that is a subset of Type2 and not of Type1.

New
Current Status:
Requires Type-EQL.

Type-Exclusivep *Type1 Type2*
Succeeds if *by definition* $Type1 \cap Type2 = \emptyset$.

New

Type-Intersectp *Type1 Type2*
Succeeds if $Type1 \cap Type2$ has some named instance, that is, a named intersection. Note that this is stronger than (NOT (Type-Exclusivep T1 T2)).

New

Matrix-Relation *Type1 Type2*
Returns the information that is stored in the matrix for the relationship between the two types.

Type-Info *Type*
Returns a list giving the relationship between the given type and every other type in the system, of the form:

((type .el type1)(type2 rel type)...)

The type you are querying can be in either the first or second slot. Note that there may be "duplicate" information, *e.g.* (Type-Info 'T-U) may return ((T-U :SUPERSET FOO) (FOO :SUBSET T-U)).

Types
Returns a list of all types known in the system.

Table 8.1 indicates the possible relationships between types:

8.11.3.1 Type Compatibility and an Example

Using the axioms above, Rhet can compute the compatibility of two terms efficiently, at least if neither term is involved in a type calculus. Types are compatible if one is a subtype of the other or if they overlap. Overlaps occur

:SUBSET	a subset relation holds between the two types.
:SUPERSET	a superset relation holds.
:NONDISJOINT	the types intersect but the overlap is not named.
:EQUAL	the types are identical.
:PARTITION	(Unimplimented as of Version 16) for entry [a b], b is a subset of a, and for all other subsets of a there is no overlap (part of a cover of a). Note that [b a] will be of type :subset, and :partition implies superset.
:DISJOINT	if [a b] do not intersect
:UNKNOWN	if the relationship is undefined, and could not be derived.
Symbol	(other than the above) the item on the list is the name of the intersection of the given types.

Table 8.1: Relationships between Rhet types in the Type Table.

in two ways: named *or* unnamed. A named overlap results from an Tname-Intersect assertion; an unnamed overlap can be implied from either Utype assertions, or a Toverlap assertion. The unification of two typed variables may result in a variable of a complex type of the form *(type1 type2) indicating the intersection of the two types. This new type is recognized in the proof as a new type. For example, suppose we have the assertions:

(8.16) (Tsubtype 'anything 'cars 'person)

(8.17) (Tsubtype 'cars 'ford 'smallcars)

(8.18) (Tsubtype 'person 'student 'worker)

(8.19) (Toverlap 'student 'worker)

(8.20) (Tname-Intersect 'pintos 'ford 'smallcars)

(8.21) (Utype 'worker 'john)

(8.22) (Utype 'student 'john) ;should imply worker, student overlap

(8.23) `[[want ?x*person ?g*ford] < [efficient ?g*ford] [wealthy ?x*person]]`

(8.24) `[[efficient ?f*smallcars] <]`

(8.25) `[[wealthy ?d*worker] <]`

We could then attempt to prove

(8.26) `[want ?f*student ?d*ford]`

and we would get

(8.27) `[want ?r*(student worker) ?u*pintos],`

pintos being a named overlap while the intersection of the types student and worker is derived by the prover.

8.12 Structured Types

Strutured types as a special form of Rhet type. They act just like normal Rhet types, but have additional properties associated with them. In a gross sense, they are very much like flavors in Lisp, but rather than each slot having a value, it is used for equality. This has advantages and disadvantages. The principle disadvantage is that a slot cannot be said to really have a "value"[40], and one entry in the equivalence class is treated by Rhet the same as any other. This means that one can't really deal with determining if a "value" has been found yet for a slot except by having your application recognize appropriate members of the equivalence class specially. The principle advantage is in fact related to this problem: we can talk about the roles of several different objects as being "equal", without knowing what their "value" is. In fact, this notion of equality is stronger than this; it is sort of a Lisp EQ vs. EQUALP distinction: some systems allow us to see if the contents of two distinct cells happen to be the same, while Rhet deals with equality on a more fundamental level: the two slots can be DEFINED to be the same. This is not merely sharing a cell, it is a logical relationship using Add-EQ that can be independantly reasoned about.

8.12.1 Defining Roles and Relations in the Type Hierarchy

Define-Subtype *Type Supertype* &Key *(Roles* (Rolename Type)*) (Relations* (Relationname Relation-Forms*)*) *Constraints Initializations*

Defines Type as a subtype of Supertype, defines the indicated type restricted roles for the new type, and asserts or notes the Constraints (any Rhet predicates, including equality constraints). In addition, Type inherits any roles from Supertype. An inherited role may be redefined only if its new type restriction is a subtype of the inherited type restriction, *e.g.*,

(8.28) (Define-Subtype 'T-ACTION 'T-U :roles '((R-ACTOR T-ANIM)))

defines T-ACTION to be a subtype of T-U, with a role R-ACTOR defined and restricted to be of type T-ANIM. This is roughly equivalent to adding:

[40] But see Set-Functional-Value.

(8.29) (Tsubtype 'T-U 'T-ACTION)

and defining f-actor by

(8.30) (Declare-FN-Type 'f-actor '(T-ACTION) 'T-ANIM).

In addition, Define-Subtype sets up some internal data structures to maintain the role inheritance in an efficient manner. Define-Subtype can also be used to declare that certain roles will be constrained, *e.g.* with the above Define-Subtype we could now assert:

```
(8.31) (Define-Subtype 'T-Self-Action 'T-Action :Roles '((R-Action-Object T-U))
                                                :Constraints '([EQ? [F-Actor ?self]
                                                                    [F-Action-Object ?self]]))
```

Note the use of the variable ?self as a stand in for the instance being created of this type. NB: While we will eventually support more than one variables in constraints, *e.g.*

```
(8.32) (Define-Subtype 'T-Foo-Action 'T-Self-Action
           :Roles '((R-Foo-Object T-U))
           :Constraints '([Foo-P [F-Foo-Object ?self]
                                 [C-Mumble [F-Actor ?self]
                                           ?z*T-Mumble-Role-2]]))
```

which would allow matching of the var to the instance the constructor builds (since it is unused), currently such constraints are unsupported. Currently, only one variable is allowed to appear in either the list of constraints or the list of initializations which will be unified with the instance created.

Initializations are a list of Rhet forms that will be proved at instance creation time. Normally, these will be builtins with side effects, such as Set-Function-Value. A single variable which can be unified with the instance must appear in the initializations, and at instance creation time, the initializations forms will each be deterministically proved with this variable bound to the instance being created. A typical use for the :Initializations option would be to set up a value cell via Set-Function-Value for roles that need to have such a thing. For example,

(8.33)
```
(Define-Functional-Subtype 'T-Eat-Lunch-Plan 'T-Plans
    :Roles '((R-Agent T-Human)
             (R-Object T-Food)
             (R-Plan-Steps T-List)
             (R-Store T-Eating-Place))
    :Initializations '(([Set-Function-Value
                         [F-Plan-Steps ?self]
                         ([C-Go [F-Agent ?self] [F-Store ?self]]
                          [C-Find [F-Agent ?self] [F-Object ?self] [F-Store ?self]]
                          [C-Take [F-Agent ?self] [F-Object ?self]])])))
```

appropriately sets up the R-Plan-Steps role for the specific type being instantiated. Note that DefRhetFun can be used at the top level to initialize the value cell based on the function itself rather than the type.

There are four cases of constraint types that need to be considered:

Foldable, Non-Monotonic which are forms that can be undone, but it is desireable to fold their effect in immediately. Rhet can tell that a predicate is Foldable and Non-Monotonic on the basis of declarations made via the DefRhetPred function. Also, this is the default type of constraint. Constraints that are foldable means that their effect may be asserted. Non-monotonic means that once they hold they do not necessarily continue to hold. Currently, Rhet will assert F/NM constraints, but will never test for them, so they are the equivalent of F/M constraints. For example, to claim that constraint `[Foo-p [R-Slot1`

?x*T-Being-Defined]] is F/NM means that the system will simply Assert this to be true, without binding the variable, *i.e.* do [Assert-Axioms [[Foo-p [R-Slot1 ?x*T-Being-Defined]] <]].

Foldable, Monotonic *e.g.* equality which once done cannot be undone, and are always valid. FC works for these. User predicates that are monotonic would work here too: once we assert them as true we no longer have to worry about them. In some sense, this is equivalent to the Initializaions field, except for a small matter of semantics. Where on initializations we would write Add-EQ, on constraints we write [EQ?], since here we are trying to describe a set of forms that should be provable, rather than a set of forms that should be asserted.

Non-Foldable, Non-Monotonic These would include bizzare user lisp functions, and other non-monotonic constraints. Their major use probably wouldn't be in an appropriate KR (at least I can't think offhand what the use would be), but could have a major procedural use for debugging. The idea would be that these are predicates that cannot be asserted (hence non-foldable), but we may want to test them whenever a change is made to the object to make sure they are true. In this sense they are similar to the Lisp ASSERT function, et. al. and could be used to detect erronious deletions or type changes that are causing subtle bugs later in the program. One can imagine, for example, a bird that must have the property CAN-FLY, which at some point in the proof is provable, but at some later point, the property is *retracted*. We may want to put a TEST (rather than a (foldable) assertion) that the content of a slot can-fly to catch this erroneous deletion and help us track the bug.

Syntactic Representation Not Finalized

Non-Foldable, Monotonic These would include certain builtins that once true will remain true, but cannot be "asserted" to be true in the sense that EQ or a user predicate could. Monotonic lispfns would come under this header, for example, as well. E.g.

(8.34) [Parent-of [F-Object ?x] [F-Actor ?x]]

At the surface, this LOOKS like something we can assert to be true. The problem is that if we know that Jim and Sue are Mary's parents, and for this instance we make the object Mary but the actor Bill, then Bill becomes one of Mary's parents! Either we have to have a rule which states that not only Jim and Sue Mary's parents, but Mary has no other parents, or we have to have declared Parent-of to be

Non-Foldable, that is, NOT something we want to ASSERT to be true, but something we want to TEST and be SURE it is true. We may not want to write the rule like this (which would confuse us with F/M), so let us instead write something like

(8.35) `[EQ? [Any ?x*t-human [Parent-of [F-Object ?y] ?x]] [F-Actor ?y]]`

which forces BC on the Parent-of test for particular choices of F-Actor. Of course, this is a case of EQ that cannot be FC'd, but must be tested (it is not foldable) since the constrained variable is not ground at instance creation time. This is a somewhat obscure declaration, we can make it easier to express by using something like the POST function, e.g.

(8.36) `[POST [Parent-of [F-Object ?x] [F-Actor ?x]]]`

could be the semantic equivalent.
Or, if we were to use the function cell of the PARENTS function, we might do:

(8.37) `[Member [F-Actor ?x] [Function-Value [Parents [F-Object ?x]]]]`

which again could be FC'd, but would be erroneous in the same way the above example was (we would be SETTING the function value instead of CHECKING it).
Another example: we may want to say that

(8.38) `[My-Predicate [F-Actor ?x] [Any ?z [Another-Predicate [F-Object ?x] ?z]]]`

Here is something else that is a perfectly valid *constraint*, but is not valid as an *assertion* (that is, the actor slot must make My-Predicate true for some other variable that some other predicate is true of). The whole idea here is that we may have *constraints* that are not *assertions*, and we should not confuse the two.
Define-Subtype also has a mechanism for setting up some standard relations. Once a relation has been defined using Define-REP-Relation, it can appear as a clause in the :Relations keyword option. Unlike

Constraints and Initializations, Relations are not processed by the usual mechanisms, but left for the user to deal with via builtin functions. Relation-Form? and Relation-List are the builtins that can be used to manipulate the defined relations for a structured type. Note that relations are defined on a type, and not on an instance; it is up to the user to decide how to use the relations, *e.g.* bind ?self to a particular instance, prove them, *etc.*.

Here is an example. First we define a structured type whose subtypes will contain an assertable STEPS relation.

(8.39)
```
(Define-Subtype 'T-Plans-With-2-Steps 'T-U
    :Roles '((Agent T-Animate))
    ;; since we know we will have two steps...
    :Constraints '([EQ? [F-AGENT ?SELF] [F-AGENT [STEPS-1 ?SELF]]]
                   [EQ? [F-AGENT ?SELF] [F-AGENT [STEPS-2 ?SELF]]])
    :Initializations '([ASSERT-RELATIONS ?SELF :STEPS]))
```

What 8.39 does for us is define a type with one role (an Agent), and says that any time we instantiate some instance of the type, "assert" the forms on the :STEPS relation. Further, constrain the agent of each of the two steps to be the agent of this instance. (We actually didn't need to do this as we will shortly see). Now define some subtype:

(8.40)
```
(DEFINE-SUBTYPE 'T-HUNT 'T-Plans-With-2-Steps
    :Relations '((:STEPS [C-GO-TO-WOODS [F-AGENT ?SELF]]
                         [C-GET-GUN [F-AGENT ?SELF]])))
```

This defines two (by default, uninterpreted) steps on the T-Hunt type, which are in fact, constructor functions. Since this is a subtype of T-Plans-With-2-Steps, we inherit it's initializations, so when we instantiate some instance of T-Hunt, we will have it's :Steps relaitons asserted. Note that since we are using constructor functions in the steps that each take one argument, the agent, and this agent is defined as the agent role of the instance, we don't really need the inherited contstraints from the parent

type, though they do point out the use of the constructed step accessors that Assert-Relations builds. Note how we have essentially constructed the same effect that our example in 8.33 did, but much more concisely and easily readable than Set-Function-Value forms buried in the initializations.

So, lets see what happens when we do (Define-Instance [Hunt-1] 'T-Hunt :Agent [Robert]):

1. We set the type of the function term [Hunt-1] to be T-Hunt.
2. Because T-Hunt is a structured type, we assert its role relations, namely [Add-EQ [F-Agent Hunt-1] [Robert]].
3. Because T-Hunt is NOT a functional type, nor is it a child of one, we do not establish a constructor function equivalent to [Hunt-1]. Had it been we would have asserted [Add-EQ [C-Hunt Robert] [Hunt-1]].
4. We run the initializations on the instance. This binds ?self appropriately, so we try to prove [Assert-Relations Hunt-1 :Steps]. Since the type of [Hunt-1] is T-Hunt, we use the appropriate relations for that type. We thus assert [Add-EQ [C-Go-To-Woods [F-Agent Hunt-1]] [Steps-1 Hunt1]] and [Add-EQ [C-Get-Gun [F-Agent Hunt-1]] [Steps-2 Hunt-1]]. The constructor functions will also be appropriately instantiated (*e.g.* for the first one we will do something like [Add-EQ [F-Agent [C-Go-To-Woods [F-Agent Hunt-1]]] [F-Agent Hunt-1]].
5. We attempt to prove or assert the constraints (that are inherited from T-Plans-With-2-Steps. Since by definition these are provably true, we do not need to add any additional information.

Note that Define-Subtype (as well as Define-Functional-Subtype, below), may only refer to functions that have already been typed, or refer to the function being typed, in the Initializations, Relations, or Constraints field.

Define-Functional-Subtype *Type Supertype* &Key *(Roles* (<Rolename Type>)*) *(Relations* (Relationname Relation-Forms*)*) Constraints Initializations*

This defines Type in the same manner as the Define-Subtype function, but in addition defines a constructor function for the type. Thus, given the definition of T-ACTION above,

(8.41) (define-functional-subtype 'T-EAT 'T-ACTION :roles '((R-OBJ T-FOOD)))

would define T-EAT to be a subtype of T-ACTION with roles R-OBJ and R-ACTOR (inherited), and would define the function f-obj for the R-OBJ role and a constructor function c-eat. This is roughly equivalent to adding:

(8.42) (Tsubtype 'T-ACTION 'T-EAT)

where the functions are defined by

(8.43) (Declare-FN-Type 'f-obj '(T-EAT) 'T-FOOD)

(8.44) (Declare-FN-Type 'c-eat '(T-FOOD T-ANIM) 'T-EAT).

although the c-eat function has the side effect of defining a new instance. The definition of f-actor for T-ACTION will apply as needed to instances of T-EAT.

Note that Define-Functional-Subtype (as well as Define-Subtype, above), may only refer to functions that have already been typed, or refer to the function currently being typed, in the Initializations, Relations, or Constraints field.

Define-Conjunction *New-Type* (Existing-Type*) &Key *:Roles* ((New-Role-Name Existing-Role-Name*)*)

New
Current Status:
To be tested

This defines New-Type to be a conjunction of the properties of the Existing-Types. Where one type may define the AGENT to be the ACTOR and another call it the AGENT, the :roles keyword allows explicitly informing RHET that these are indeed the same slot and which name should be used for access to it from the conjunction type.

Define-REP-Relation *Relation-Keyword* &Optional (Inherit-type :Inherit) New

Define Relation-Keyword (a keyword) to be an allowable relation for future calls on Define-Subtype and Define-Functional-Subtype. If Inherit-type is :Inherit, if this relation is given when defining a subtype, it is concatenated to the inherited definition. If it is :Redefine, it takes priority over the inherited one (the inherited definition is only used if there is no local definition). If it is :Local, the relations are not inherited at all from parent types. Note that REP-Relations must be defined before any use, and redefinition will only effect definition that occur afterwards.

An example of Define-REP-Relation's usage is given above. Rhet predefines the relations :Steps, :Preconditions, and :Effects as a convienience to the user.

The Rhet system provides a convenient abbreviated form for defining instances of structured types.

Define-Instance *Instance Type* &Rest <<*Rolename*> <*Value*>>* &Key *Context*

This defines Instance to be an Itype of Type and defines the indicated roles of Instance to have the indicated values. Instance and each Value are Rhet Forms. A context keyword and value may be supplied, but they must appear last in the argument list.

For example,

(8.45) (Define-Instance [E1] 'T-EAT 'R-OBJ [F1] 'R-ACTOR [JOE])

is equivalent to adding

(8.46) (Itype 'T-EAT [E1])

(8.47) (Add-Role [E1] 'R-OBJ [F1])

(8.48) (Add-Role [E1] 'R-ACTOR [JOE])[41]

with the main difference being that Define-Instance is a Lisp function (only), and the others are (potentially) horn clauses that could be added during a proof.

Either way of asserting this information will cause the following equalities to be derived:

(8.49) (Add-EQ [c-eat F1 JOE] [E1])

(8.50) (Add-EQ [f-obj E1] [F1])

(8.51) (Add-EQ [f-actor E1] [JOE])

Further, note that the builtin Add-Role, is also available as a Lisp function:

Add-Role *<Instance>* &Rest *<<Role-Name> <Value>>** &Key *Context* Was Role

Adds that object <Instance> has role <Role-Name> with value <value>. All arguments must be fully grounded. Note that Instance and each Value are expected to be Rhet Forms. Unlike the builtin, the lisp function will also take an optional Context keyword argument pair. If supplied, it must be last.

8.12.2 Retrieving Structured Type Information

The Rhet system provides a general facility for providing information about any object defined. This is provided by the following functions:

Retrieve-Def *Object* Changed

which returns a description of the object as per the formats in table 8.2. Note that while the table shows a list being returned, Retrieve-Def actually returns multiple values.

New

Rep-Structures

Return a list of all structured types defined.

New
Current Status:
To be tested

Classify *Type :Roles* ((Role-Name Role-Type))*

This will return the (simple) types that are subsumed by Type and subsume types which have roles at least as general as the ones explicitly given. As an example, assume we have the following in our KB: The Action type has the role R-Agent which is required to be a Person; the ObjAction type takes the roles R-Agent restricted to Person, and R-Obj restricted to T-U, and the Drive type with roles R-agent restricted to be of type Person, and R-Obj restricted to be of type Automobile. In this case were we to do

(8.52) (Classify 'Action :Roles '((R-Agent Person)(R-Object PhysObj)))

we know the types subsumed by Action are ObjAction and Drive, but since ObjAction is more general than our query (specifically the R-Object role defined to be T-U which would subsume our own R-Object's type), we returns the type Drive since each role our query subsumes the defined roles of Drive.

New
Current Status:
To be tested

Subsumes *Type* (Role-Name Role-Type)*

This succeeds if the given roles and value restrictions all subsume those of Type.

In general, an unknown type is considered to subsume a known type if:

1. The preliminary guess as to the type of the unknown subsumes the type of the known.
2. The roles of the known are a subset of those of the unknown.
3. Each of the role restrictions on each of the roles of the known type are subtypes of those of the unknown.

For example, given the definition of T-OBJ-ACTION above, (Retrieve-Def 'T-OBJ-ACTION) would return:

(8.53) (TYPENAME (T-ACTION) (R-OBJ R-ACTOR) (T-PHYS-OBJ T-ANIM))

Type of Object	Return Values:	(as a list, actually uses CL multiple values)
Type	(:TYPE-NAME	<list/immediate supertypes> <list/roles defined> <list/role type restrict.> <list/foldable non-monotonic constraints> <list/foldable monotonic constraints> <list/non-foldable non-monotonic constraints> <list/non-foldable monotonic constraints> <list/initializations> ((<relation-name> <relation-form>*)*))
Rolename	(:ROLE-NAME	<list of types using that role>)
Function Name	(:FUNCTION-NAME	((<type restrictions on args> <type of result>)*)
Relation	(:RELATION-NAME	<list of types using that relation>)
Free Variable	(:VARIABLE	<type restriction>)
Constant	(:CONSTANT	<Type> (<role> <value>)*)
Func. w/ unbound vars	(:FUNCTION	<type> (<role> <value>)*)
Anything Else	(:UNKNOWN	)

Table 8.2: Object Descriptions Returned by Retrieve-Def

For another example, if A is an instance of T-OBJ OBJ-ACTION with the R-OBJ role set to O1 and R-ACTOR set to (f-actor A2), then (Retrieve-Def [A]) would return:

(8.54) (CONSTANT T-OBJ-ACTION (R-OBJ [O1]) (R-ACTOR [f-actor A2])).

Finally, given a function containing unbound variables, Retrieve-Def will return as much information as it can derive using the basic format for constants, but differing in the first atom, *i.e.*, it returns

(8.55) (FUNCTION <type> (<role> <value>)*)

as mentioned in table 8.2.

For example, given the assertions of 8.28, 8.41, and 8.45, we would retrieve the following:

```
Rhet-> (retrieve-def 'T-EAT)
(:TYPENAME (TYPEKB::T-ACTION) (R-ACTOR R-OBJ) (T-ANIM T-FOOD) NIL NIL NIL NIL NIL NIL)
Rhet-> (retrieve-def 'R-OBJ)
(:ROLE-NAME (TYPEKB::T-EAT))
Rhet-> (retrieve-def 'R-ACTOR)
(:ROLE-NAME (TYPEKB::T-EAT TYPEKE::T-ACTION))
Rhet-> (retrieve-def 'f-obj)
(:FUNCTION-NAME (((T-ANYTHING) . *T-ANYTHING) ((*T-EAT) . *T-FOOD)))
Rhet-> (retrieve-def 'c-eat)
(:FUNCTION-NAME (((*T-ANIM *T-FOOD) . T-EAT)))
Rhet-> (retrieve-def ?x*T-EAT)
(:VARIABLE T-EAT)
```

```
Rhet-> (retrieve-def [E1])
(:CONSTANT T-EAT (R-ACTOR [JOE] R-OBJ [F1]))
Rhet-> (retrieve-def [f-obj E1])
(:CONSTANT T-FOOD NIL)
Rhet-> (retrieve-def '[c-eat JACK [f-obj E1] ])
(:Function T-EAT (:Dont-calc-roles-for-constructors-yet))
Rhet-> (retrieve-def '[c-eat JACK ?x*FOOD ])
(FUNCTION T-EAT (:Dont-calc-roles-for-constructors-yet))
Rhet-> (define-instance [E2] 'T-EAT 'R-ACTOR [JACK])
[E2]
Rhet-> (retrieve-def [E2])
(:CONSTANT T-EAT (R-ACTOR [JACK] R-OBJ [f-obj E2]))
Rhet->
```